Automatic Generation of Safe Handlers for Multi-Task Systems

Eric Rutten, Hervé Marchand

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automatic Generation of Safe Handlers for Multi-Task Systems

Éric Rutten — Hervé Marchand

N° 5345

21st October 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5345--FR+ENG

RHÔNE-ALPES

Automatic Generation of Safe Handlers for Multi-Task Systems

Éric Rutten*, Hervé Marchand†

Thème 1 — Réseaux et systèmes
Projets POP ART et Vertecs

Rapport de recherche n° 5345 — 21st October 2004 — 44 pages

Abstract: We are interested in the programming of real-time control systems, such as in robotic, automotive or avionic systems. They are designed with multiple tasks, each with multiple modes. It is complex to design task handlers that control the switching of activities in order to insure safety properties of the global system. We propose a model of tasks in terms of transition systems, designed especially with the purpose of applying existing discrete controller synthesis techniques. This provides us with a systematic methodology, for the automatic generation of safe task handlers, with the support of synchronous languages and associated tools for compilation and formal computation.

Key-words: Real-time systems, safe design, discrete control synthesis, synchronous programming.

* Eric.Rutten@inrialpes.fr, www.inrialpes.fr/pop-art
† Herve.Marchand@irisa.fr, www.irisa.fr/vertecs

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Génération automatique de gestionnaires sûrs de systèmes multi-tâches

Résumé : Nous nous intéressons à la programmation de systèmes temps-réel tels qu'on les trouve dans les systèmes robotiques, automobiles ou avioniques. Il sont conçus avec de multiples tâches, chacune dotée de modes multiples. Il est complexe de concevoir les gestionnaires de tâches qui contrôlent la commutation des activités de façon à assurer des propriétés de sûreté sur le système global. nous proposons a modèles de tâches en termes de systèmes de transition, conçus spécialement dans le but d'appliquer des techniques existantes de synthèse de contrôleurs discrets. Ceci nous donne une méthodologie systématique pour la génération automatique de gestionnaires de tâches sûrs, sur la base des langages synchrones et des outils associés de compilation et calcul formel.

Mots-clés : Systèmes temps-réel, conception sûre, synthèse de contrôleurs discrets, programmation synchrone.

1 Multi-task systems

1.1 Real-time control systems

We are interested in the design of real-time control systems, such as in robotic, automotive or avionic systems, where the controller implements control laws. This involves different aspects like e.g., device (sensor and actuator) management, numerical computation, and the discrete, event-based switching between modes of activity. Such systems are designed with a number of tasks, executed in a cyclic way, each computing a closed loop control. An example of such a control system architecture is ORCCAD [5]. Each task can have different modes, which can be different phases (initialization, nominal, exception handling), or versions implementing the same functionality with different levels of quality (e.g., computation approximation), and cost (e.g., computation time, energy, memory, bandwidth of communication, side-effects on the environment) [8, 28, 20]. Elaborate control systems offer multiple functionalities, managed by a number of such tasks, which can be invoked in sequence or parallel. They may then have to share processors or devices, and to be managed w.r.t. their interactions. Instances of such systems in applications are:

- control systems (e.g. robotics) where tasks implement the computation of a control law or function, in a cyclic way, reading sensor input, and producing actuator command output.

  A notion of quality of service in these systems is that the numerical computations involved, e.g., matrix computation for kinematic model update, can be performed with different levels of accuracy, by making approximations e.g., by limiting the development of series, or avoiding the computation of terms that have a negligible value under some conditions [20]. The control laws can also differ by the way energy is consumed by actuators or side-effects on the environment.

- telecommunication systems and portable devices like cellular telephones present many functionalities; e.g., signal processing as in voice recognition or encodings for transmission, can have different implementations according to the way energy is spent, or volume of data and use of bandwidth.

- automobile embedded equipments, in every of their activation configurations, have a given energy consumption: this latter has to be bounded, because of the battery, so that the autonomy of the car is not jeopardized. At the same time, priority tasks (e.g., related to safety) have to be respected.

It is complex to handle tasks and to control the switching of modes in order to insure properties characterizing the safety of the control systems. This safety is often critical in the framework of transportation systems, energy production and, in general, in all systems with a strong and possibly hazardous interaction with people or the environment. The properties to be ensured can concern the safe access to resources, according to particular constraints of sequencing or exclusions, or managing more quantitative aspects, like bounding cost while maximizing quality (i.e., limiting degradation). Fault tolerance in these systems can be seen as switching between configurations, upon the occurrence of failure of e.g., resources, while maintaining a certain level of functionality, possibly in degraded modes.

All this suggests that in a multi-task system, the control of activations and of mode switching, in order to be safe, must restrict the system within allowed combinations, with respect to statical or dynamical conditions.

1.2 Task handlers in a layered architecture

Such systems present various aspects and layers: low-level interfaces with the controlled physical system, resources (computation, communication, or controlled physical system), control tasks (with possibly different versions w.r.t. performance), and, on top of all this, applications sequencing such tasks in order to fulfill some functionality. Figure 1 depicts a general architecture of the considered systems [2], which this paper considers in the more particular case of control systems and tasks. In this framework, the library of control tasks is managed by a task handler, that receives requests from the application, and monitors execution in relation with the physical system (in particular, the termination of tasks). Resources managed by the task handler are of two kinds:

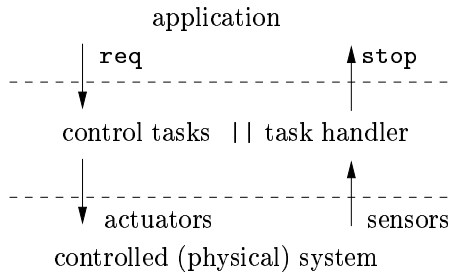


Figure 1: The considered general system architecture.

- physical resources in the controlled system: actuators (which should typically not be controlled by two different tasks, or should always be under control of some task, or should be used only under conditions concerning their physical interaction with other parts of the system), sensors (which might require to be used with particular parameter settings making them unsharable by several tasks, or which might involve some control themselves w.r.t., e.g., orientation), the way they consume energy, or the performance or quality with which they can achieve their function, ...
- computing and communication resources for the tasks: processor(s) (which have to perform cyclic computation of control within a bounded period, which can have different characteristics), memory (which can be bounded too), communications (which follow a given topology), bandwidth, ...

All these aspects involve interactions and constraints to be managed by the task handling controller. The complexity of the resulting system calls for some structuring, and assistance in the design of controllers. We are looking for ways of separating concerns between local, device-dependent aspects of control, and more global, system-level interactions, while keeping the application-independent aspects as circumscribed and re-usable as possible.

1.3 Our approach

We want to assist in the design of safe task handlers, by proposing a method which automates their generation. Our purpose is practical, and the formalization we give of the problem is meant to be hidden from a user, who will manipulate essentially a few particular but meaningful model and properties patterns.

At the level of abstraction of the task handler, we need to model a discrete behavior, characterized by the discrete states of the resources (e.g., occupied or free) and of the task (active or not, in

which mode, ...), and also possibly of the environment or of some external user interaction. A notion of global state can be derived from local states. The dynamical behavior of the system, observed from that point of view, proceeds from state to state, following transitions that occur typically upon the occurrence of significant events, like requests from a user or application, sensor events, ... Therefore, we will use a formalization in terms of finite state automata, with labeled transitions, to model the systems under study.

The tasks handler has to receive and handle requests from the application, while caring for properties of the computation and physical resources, to be satisfied. Properties generic to control systems are of the following kinds:

- for the physical resources (typically actuators): mutual exclusion of control w.r.t. actuators, permanent control of each actuator, logical constraints between actions corresponding to physical aspects (e.g., cooling action between two temperature-intensive actions), ...
- for the computing resources (processor, memory): boundedness of the global cycle time (for a time-sharing execution), switching between versions or modes of the control tasks (with different characteristics of cost in time, energy, memory, quality, ...)

In order to be able to ensure these properties, we have to design the local controllers of the tasks in such a way that they offer the appropriate control states and events to represent the possible behaviors, and to enable the building of a correct global behavior by appropriately synchronizing them. The control tasks must be structured and equipped so that properties can be enforced. They have a local behavior that can be described as a finite state machine or automaton, representing typically:

- activity state: in reaction to invocation, requests from the application level, and reception of termination signals, a task can be active or not;
- handling of requests: the model describes different responses, whether they can be delayed or not, and if they are, whether several can be memorized and queued, whether they can be forced or not;
- activity modes and commutations: when the same task can be performed according to different versions of an algorithm, or configurations of an execution platform, the automaton describes them, as well as the way control can be switched from the one to other.

Different patterns for tasks can be imagined, corresponding to different intended behaviors. The global behavior of the system is described by the composition of all such tasks. Controlling these global behaviors in such a way that the properties mentioned before are satisfied is our motivation.

Given the context explained above, the problem we address here is the safe handling of actions of control systems, given a general architecture characterized by a set of tasks, and a set of properties to ensure, concerning their logical constraints and some more quantitative aspects.

Our approach consists of:

- building an automaton-based model of the possible behaviors of the set of tasks, and distinguishing, in the conditions labeling the transitions, between events that can be controlled by the handler, and the others, uncontrollable ones;
- formulating the properties on the basis of this model, in terms of state to be reached or avoided, or of sequences to be imposed or forbidden;

- automating the building of the handler by using discrete controller synthesis; it consists of generating the functions which give, for a current state and uncontrollable events, the set of controllable events such that the properties are satisfied.

1.4 Related work

The work presented in this paper unifies first results obtained in more particular contexts. Particularly, the class of tasks constituting the systems we consider takes its inspiration in robot control systems, where a control law, designed following techniques of continuous control theory, is executed cyclically. Such computations can be controlled according to given events and states, control patterns, as proposed in e.g. the ORCCAD robot programming environment [5]. A complete system hosts a number of such tasks, which can be activated in complex sequences. Most formal methods approaches applied on such models concern verification. In our approach we explore the use of discerete controller synthesis, as a more constructive technique.

A first approach inspired by robotic systems was proposed, using a teleoperation application as illustration [30, 31]. Another approach considered tasks with multiple modes or versions, and their characterization by weights associated to states, and used in optimal synthesis [25]. This paper proposes a unified formalization, and complements in the model. Such an automated construction of the task handler can be generalized as a way of compiling a controller from a mixed language [2]. Such a compilation is not classical, in that it makes use not only of a formal model of behaviors of the compiled program, which is already the case in the compilation of synchronous languages concerning invariants [11], but of a model of its dynamics [17].

Concerning applications of discrete controller synthesis, related approaches working with models of task systems can be found, often using timed models and synthesis techniques [18]. The orientation then is more towards the synthesis of application specific schedulings for real-time tasks [15]. We concentrate on rather logical and discrete aspects (even when considering static weights), where we feel the algorithmic complexity costs are much less, and allow for a greater scalability potential. In a component-based design setting [3], synthesis can be used to compute the allowed interactions, given constraints and properties to be enforced, of components seen through an interface characterizing their input and output conditions; it produces a form of partial protocol, leaving some non-determinism to be resolve by an on-line scheduler in a fair manner.

1.5 Outline

In the following sections, we will first present the formal framework of our approach in Section 2 as well as an implementation based upon synchronous tools. We then describe the patterns for task behavior in Section 3.2, as well as the models for typical properties in Section 3.3 and 3.4. Finally, a case study illustrates the approach in Section 4.

2 Automata and Controller Synthesis

The basic models are discrete-event systems, and can be formulated as, e.g., formal languages, Petri nets, or finite state machine [6]. In the latter case, states correspond for example to given configurations of the robot system, and/or of the activation status of tasks controlling it. Transitions correspond to the occurrence of, e.g., events, commands, thresholds. They are labels on the transitions, which can involve conditions on their fireability. In the sequel, we shall consider transition systems, in which events can occur simultaneously. Hence a transition between two consecutive

states can be labeled by a vector or conjunction of events. This is related to the synchronous composition we will be using.

Given a property involving states or events, verification consists in determining, whether or not they are true for a given automaton, representing the possible behaviors of a program or system [4]. Differently, discrete control synthesis consists of determining the constraints that make the resulting automaton satisfy the property, by inhibiting the transitions which would lead to its violation. Hence, the synthesis of a controller consists of automatically computing the *controller*: a relation that, given a state and uncontrollable events, gives the value of controlled events such that only transitions respecting the objectives can be taken (in other words: contradicting behaviors are inhibited), as illustrated in Figure 5 in the case of a deterministic controller. This produces a constrained model, i.e. model and controller together satisfy the property. It involves restrictions on admissible initial states, as well as on transitions.

In the following, we give definitions inspired by [16, 26, 2], and introduce graphical notations which will be used for convenience further.

2.1 Synchronous Automata

Given a set of boolean variables $\mathcal{V} = \{\mathcal{V}_1, \cdots, \mathcal{V}_n\}$, we first define a *valuation* of the set of variables $\mathcal{V}$ as a function: $val : \mathcal{V} \longrightarrow \{\texttt{true}, \texttt{false}\}$. The valuation val assigns to each variable in $\mathcal{V}$ a value, either true or false. The set of valuations of $\mathcal{V}$ is noted $\mathcal{V}al(\mathcal{V})$. Given a boolean expression $B(\mathcal{V})$ and $v \in \mathcal{V}al(\mathcal{V})$, we denote by $B(v)$ the predicate B valuated according to the values of $\mathcal{V}$. Moreover, we denote by B^+ the set of variables that appear as positive elements in the predicate B, i.e. $B^+ = \{\mathcal{V}_i \in \mathcal{V} \mid \mathcal{V}_i \wedge B(\mathcal{V}) = B(\mathcal{V})\}$. Respectively, we denote by B^- the set of variables that appear as negative elements in the predicate B, i.e. $B^- = \{\mathcal{V}_i \in \mathcal{V} \mid \neg\mathcal{V}_i \wedge B(\mathcal{V}) = B(\mathcal{V})\}$.

Now, the system on which control will be applied is modeled by a *Synchronous automaton*, labeled by expressions on simultaneous events, that is defined as follows:

Definition 1 *A synchronous automaton $\mathcal{A}$ is the tuple $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ such that:*

1. *$\mathcal{Q}$ is a finite set of* states*; $\mathcal{Q}_{init} \in \mathcal{Q}$ is a set of* initial states*;*
2. *$\mathcal{V}$ and $\mathcal{O}$ are the sets of Boolean Input and Output variables.*
3. *$\mathcal{C}_{init} : \mathcal{Q}_{init} \longrightarrow \mathcal{B}ool(\mathcal{V})$ is the* initial conditions *attached to initial states; $\mathcal{B}ool(\mathcal{V})$ is the set of boolean expressions over the set of variables $\mathcal{V}$: it is generated by the grammar: $b ::= x \| b$ and $b \| not\ b$, where $x \in \mathcal{I}$;*
4. *$\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{V}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of* transitions*. For $t = (q, B(\mathcal{V}), O, q')$, q, q' are the source and the target states, $B(\mathcal{V}) \in \mathcal{B}ool(\mathcal{V})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered.*

Now, for a state q and a valuation $v \in \mathcal{V}al(\mathcal{V})$ of the variables, we say that v is *admissible* in q whenever there exists a transition $(q, B(\mathcal{V}), O, q') \in \mathcal{T}$ such that $B(v) = true$; at the same time, the set of variables O is emitted. The transition is also said to be admissible.

Example 1 *Figure 2 illustrates this definition, with a graphical syntax which will be used in the remainder of the paper.*
We see states $\{A, B, C\}$, with initial states $\{A, B\}$, the input variables are $\{a, b, c1\}$ and the set of output variables is reduced to the singleton $\{c\}$, initial conditions $\mathcal{C}_{init}(A) =$ `not c1` *and $\mathcal{C}_{init}(B) =$* `c1`*, and transitions indicated by arrows: each has a source and a sink state, and a label with a*

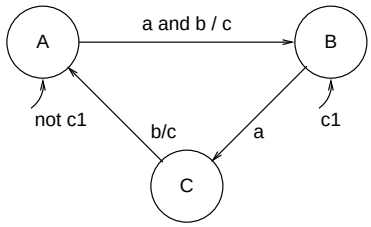


Figure 2: An example automaton.

Boolean expression on variables giving its triggering condition, and emissions. Implicit transitions, not represented by arrows, are the ones going from each state to itself when no condition labeling an outgoing transition is true. ◇

Definition 2 *A synchronous automaton* $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ *is* deterministic *whenever*

$$\forall q_{init}, q'_{init} \in \mathcal{Q}_{init},\ \forall v \in \mathcal{V}al(\mathcal{V})\ .\ \big(\mathcal{C}_{init}(q_{init})(v) \wedge \mathcal{C}_{init}(q'_{init})(v) \neq \mathit{false} \implies q_{init} = q'_{init}\big)$$

$$\forall q, q', q'' \in \mathcal{Q},\ \forall v \in \mathcal{V}al(\mathcal{V})\ .\ \forall (q, B, O, q'), (q, B', O', q'') \in \mathcal{T}\ . \quad \big(B(v) \wedge B'(v) \neq \mathit{false} \implies O = O'\ \wedge\ q' = q''\big)\ . \tag{1}$$

•

Hence, a synchronous automaton is deterministic whenever given an initial valuation v of the variables $\mathcal{V}$, there is only one valid initial state and given a state and a valuation of the variables, then only one destination state can be reached and the outputs are identical.

Definition 3 *A synchronous automaton* $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ *is* reactive *whenever*

$$\forall v \in \mathcal{V}al(\mathcal{V})\ .\ \bigvee_{q_{init} \in \mathcal{Q}_{init}} \mathcal{C}_{init}(q_{init})(v) = \mathit{true} \tag{2}$$

$$\forall q \in \mathcal{Q}, \forall v \in \mathcal{V}al(\mathcal{V})\ .\ \bigvee_{q' \in \mathcal{Q}.(q,B,O,q') \in \mathcal{T}} B(v) = \mathit{true}\ . \tag{3}$$

A synchronous automaton is reactive whenever for all states, whatever the valuation of the variables, a transition can be triggered (possibly from q to q itself).

The semantics of an automaton. $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{\text{init}}, \mathcal{C}_{\text{init}}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ is the following. Assume $\mathcal{A}$ is initialized in state $q_{\text{init}} = q_o \in \mathcal{Q}_{\text{init}}$ with valuation v_o such that $\mathcal{C}_{\text{init}}(q_{\text{init}})(v_o) = \mathit{true}$. Further, with the valuation v_1, $\mathcal{A}$ evolves into state q_1 and emits O_1 such that $(q_o, B_1, O_1, q_1) \in \mathcal{T}$ and $B_1(v_1) = \mathit{true}$. Assume now that the system has evolved so far into state q_i (i.e. after the reception of i variable valuations), then upon the reception v_{i+1}, the system will evolve into state q_{i+1} and emit O_{i+1}, such that $(q_i, B_{i+1}, O_{i+1}, q_{i+1}) \in \mathcal{T}$ is an admissible transition in the synchronous automaton $\mathcal{A}$ and $B_{i+1}(v_{i+1}) = \mathit{true}$ (i.e. B_{i+1} valuates to *true* w.r.t. the valuation v_{i+1}).

Now given a synchronous automaton $\mathcal{A}$, its behavior is characterized in terms of traces depending on variables. Let $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{\text{init}}, \mathcal{C}_{\text{init}}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ and $v_0, v_1, v_2, ..., v_n, ... \in \mathcal{V}al(\mathcal{V})$ be an infinite sequence of valuations of the variables $\mathcal{V}$. A *trace* is a sequence of tuples $t = \{(v_i, q_i, O_i)\}_{i \geq 0}$ where $q_i \in \mathcal{Q}$ are states such that

$$q_o \in \mathcal{Q}_{\text{init}} \wedge \mathcal{C}_{\text{init}}(q_o)(v_0) = \mathit{true} \wedge O_0 = \emptyset \text{ and } \forall i, \exists (q_i, B_{i+1}, O_{i+1}, q_{i+1}) \in \mathcal{T}\ \wedge\ B_{i+1}(v_{i+1}) = \mathit{true} \tag{4}$$

We note $Trace(\mathcal{A})$ the set of all traces of $\mathcal{A}$. For $q \in \mathcal{Q}$, we note *Sub-trace*$(\mathcal{A}, q)$ the set of sub-traces of $\mathcal{A}$ that begin from q:

$$\begin{aligned} \textit{Sub-trace}(\mathcal{A}, q) = \big\{ t_q = \{(v_i, q_i, O_i)\}_{i \geq o} \mid q_o = q \wedge \\ \forall i \geq 0,\ (q_i, B_{i+1}, O_{i+1}, q_{i+1}) \in \mathcal{T} \ \wedge \ B_{i+1}(v_{i+1}) = true \big\} \end{aligned} \tag{5}$$

2.1.1 Composition Operators

Let $\mathcal{A}_i = (\mathcal{Q}_i,\ \mathcal{Q}_{\text{init}\,i}, \mathcal{C}_{\text{init}\,i}, \mathcal{V}_i, \mathcal{O}_i, \mathcal{T}_i)$, for $i = 1, 2$ be two automata. We want to define the operation that describes the parallel composition, called the synchronous product. This consists in connecting the output variables of $\mathcal{A}_1$ to the input variables of $\mathcal{A}_2$ whenever they have the same name and vice-versa. Our composition will also perform a form of encapsulation, in the sense that the inputs of each automaton which are outputs of the other will be encapsulated i.e., not considered to be inputs of the composition; however they remain visible as outputs. Note that we must have $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$. The parallel composition will be denoted $\mathcal{A}_1 \parallel \mathcal{A}_2$. All the output variables remain visible in the composition, however, the input variables of $\mathcal{A}_1 \parallel \mathcal{A}_2$ are given by $(\mathcal{V}_1 \cup \mathcal{V}_2) \setminus (\mathcal{O}_1 \cup \mathcal{O}_2)$ because, given an output of $\mathcal{A}_2$, an input of $\mathcal{A}_1$ can be defined in the composition (same holds for the input of $\mathcal{A}_2$). Let us now formally define the *synchronous product* between two automata.

Definition 4 *The* **synchronous product** *of two automata $\mathcal{A}_i = (\mathcal{Q}_i,\ \mathcal{Q}_{initi}, \mathcal{C}_{initi}, , \mathcal{V}_i, \mathcal{O}_i, \mathcal{T}_i)$, for $i = 1, 2$ is the automaton $\mathcal{A}_1 \parallel \mathcal{A}_2 = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ defined by:*

- $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$ *(Cartesian product on sets);* $\mathcal{Q}_{init} = \mathcal{Q}_{init1} \times \mathcal{Q}_{init2}$*;*
- $\mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2$ *and* $\mathcal{V} = (\mathcal{V}_1 \cup \mathcal{V}_2) \setminus \mathcal{O}$ *and we denote by* $\Gamma = (\mathcal{O}_1 \cup \mathcal{O}_2) \cap (\mathcal{V}_1 \cup \mathcal{V}_2)$ *the set of variables that were both input and output variables;*
- $\mathcal{C}_{init}(\langle q_1, q_2 \rangle) = \mathcal{C}_{init1}(q_1) \wedge \mathcal{C}_{init2}(q_2)\ \forall q_i \in \mathcal{Q}_{initi},\ i = 1, 2$*;*
- $\mathcal{T}$ *is defined by*

$$\left\{ \begin{array}{l} (q_i, B_i(\mathcal{V}_i), O_i, q_i') \in \mathcal{T}_i,\ \forall i = 1, 2 \\ (B_1 \wedge B_2)^+ \cap \Gamma \subseteq \mathcal{O} \\ (B_1 \wedge B_2)^- \cap \Gamma \cap \mathcal{O} = \emptyset \end{array} \right. \Leftrightarrow (\langle q_1, q_2 \rangle, \exists \Gamma (B_1(\mathcal{V}_1) \wedge B_2(\mathcal{V}_2)), O_1 \cup O_2, \langle q_1', q_2' \rangle) \in \mathcal{T}$$

$t = (\langle q_1, q_2 \rangle, B_1(\mathcal{V}_1) \wedge B_2(\mathcal{V}_2), \langle q_1', q_2' \rangle) \in \mathcal{T}$ *iff* $\forall i = 1, 2\ .\ (q_i, B_i(\mathcal{V}_i), q_i') \in \mathcal{T}_i\ .$ •

The conditions on transitions stipulate that shared positive elements should be outputs, while shared negative elements should not be outputs i.e., both parties agree on the presence of shared elements. The synchronous product of synchronous automata is both associative and commutative.

Example 2 *Figure 3 illustrates the synchronous composition.*
The left part shows the graphical syntax : inclusion of two automata in a round-cornered box, with separation by a dashed line. The right part shows the resulting automaton, with notably the fact that synchronous composition makes for transitions simultaneously in both automata. ◇

Next we define the Hierarchical Composition. Given a synchronous automaton $\mathcal{A}_b$ and a state of $\mathcal{A}_b$, q_r the idea is to refine the behavior of q_r by means of another synchronous automaton $\mathcal{A}_r$. To simplify the definition, we assume that the top-level and the low level do not share variables (i.e. $(\mathcal{V}_b \cup \mathcal{O}_b) \cap (\mathcal{V}_r \cup \mathcal{O}_r) = \emptyset$).

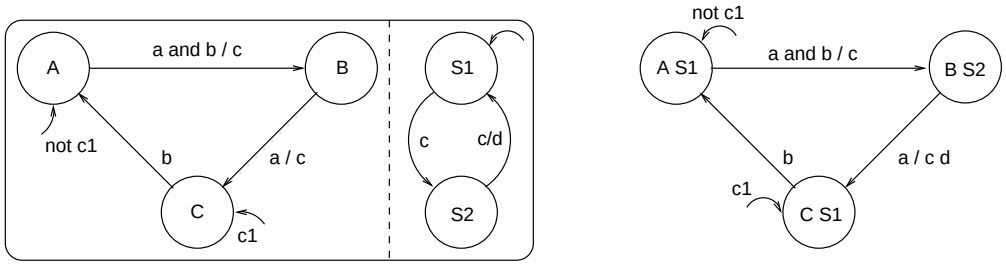


Figure 3: An example of synchronous product (left), with the resulting automaton (right).

Definition 5 (Hierarchical Composition) *Let* $\mathcal{A}_b = (\mathcal{Q}_b, \mathcal{Q}_{initb}, \mathcal{C}_{initb}, \mathcal{V}_b, \mathcal{O}_b, \mathcal{T}_b)$ *be a* basis *automaton, let* $q_r \in \mathcal{Q}_b$ *be one of its states to be refined, and let* $\mathcal{A}_r = (\mathcal{Q}_r, \mathcal{Q}_{initr}, \mathcal{C}_{initr}, \mathcal{V}_r, \mathcal{O}_r, \mathcal{T}_r)$ *be the* refinement *automaton used to refine* q_r*. The result of this refinement is the automaton* $\mathcal{A}_b \triangleright (q_r, \mathcal{A}_r) = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{T}, \mathcal{V})$ *where*

- $\mathcal{Q} = \mathcal{Q}_b \setminus \{q_r\} \bigcup \{q_r.q \mid q \in \mathcal{Q}_r\}$;
- *If* $q_r \in \mathcal{Q}_{initb}$, *then* $\forall q_{initr} \in \mathcal{Q}_{initr}$, $q_r.q_{initr} \in \mathcal{Q}_{init}$, *else* $\mathcal{Q}_{initb} = \mathcal{Q}_{init}$;
- $\mathcal{V} = \mathcal{V}_b \cup \mathcal{V}_r$ *and* $\mathcal{O} = \mathcal{O}_b \cup \mathcal{O}_r$;
- $t = (q_1, B, O, q_2) \in \mathcal{T}$ *iff*
  (1) $\big(q_1, q_2 \in \mathcal{Q}_b \setminus \{q_r\} \wedge t \in \mathcal{T}_b\big) \bigvee$
  (2) $\big(q_1 \in \mathcal{Q}_b \wedge q_2 = q_r.q_{initr} \wedge (q_1, B', O, q_r) \in \mathcal{T}_b \quad \wedge B = B' \wedge \mathcal{C}_{initr}(q_{initr})\big) \bigvee$
  (3) $\big(\exists q, q' \in \mathcal{Q}_r \ s.t. \ q_1 = q_r.q \wedge q_2 \in \mathcal{Q}_b \setminus \{q_r\} \wedge$
  $(q_r, B', O', q_2) \in \mathcal{T}_b \wedge (q, B'', O'', q') \in \mathcal{T}_r \ \wedge \ B = B' \wedge B'' \wedge O = O' \cup O''\big) \bigvee$
  (4) $\big(\exists q \in \mathcal{Q}_r \ s.t. \ q_1 = q_r.q \wedge \exists q' \in \mathcal{Q}_r \ s.t. \ q_2 = q_r.q' \wedge$
  $\exists (q, B', O, q') \in \mathcal{T}_r \wedge B = B' \wedge \neg \vee_{(q_r, B'', q'') \in \mathcal{T}_b} B''\big)$. •

The last bullet depicts when a transition exists in the composed automaton: (1) describes the transitions outside of the refined state q_r, (2) considers the transitions entering q_r whereas (3) deals with the transitions leaving q_r. (4) describes the transitions inside q_r: such a transition exists only if no transition leaving q_r in the basis automaton is allowed at the same time.
Applying this to each state gives the whole hierarchical operator: from the basis automaton $\mathcal{A}_b$, each state $q_i \in \mathcal{Q}_b$ is refined into the automaton $\mathcal{A}_i$. The result is the synchronous automaton noted $\mathcal{A}_b \triangleright (\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_{|\mathcal{Q}_b|})$ and computed by $(... ((\mathcal{A}_b \triangleright (q_1, \mathcal{A}_1)) \triangleright (q_2, \mathcal{A}_2)) ... \triangleright (q_{|\mathcal{Q}_b|}, \mathcal{A}_{|\mathcal{Q}_b|})$.

Example 3 *Figure 4 illustrates the hierarchical composition.*
The left part shows the graphical syntax : inclusion of the refinement automaton in the round-cornered box of the refined state. The right part shows the resulting automaton. ◇

2.1.2 Temporal Properties on Automata

Such transitions systems can have properties related to the reachability of some subset of the state space, or to the existence of paths along which a certain sequence of events exists. They can concern invariants on the states themselves (i.e., the variables of which the valuation defines a state), or the paths that can be taken in the transition system from state to state, etc.

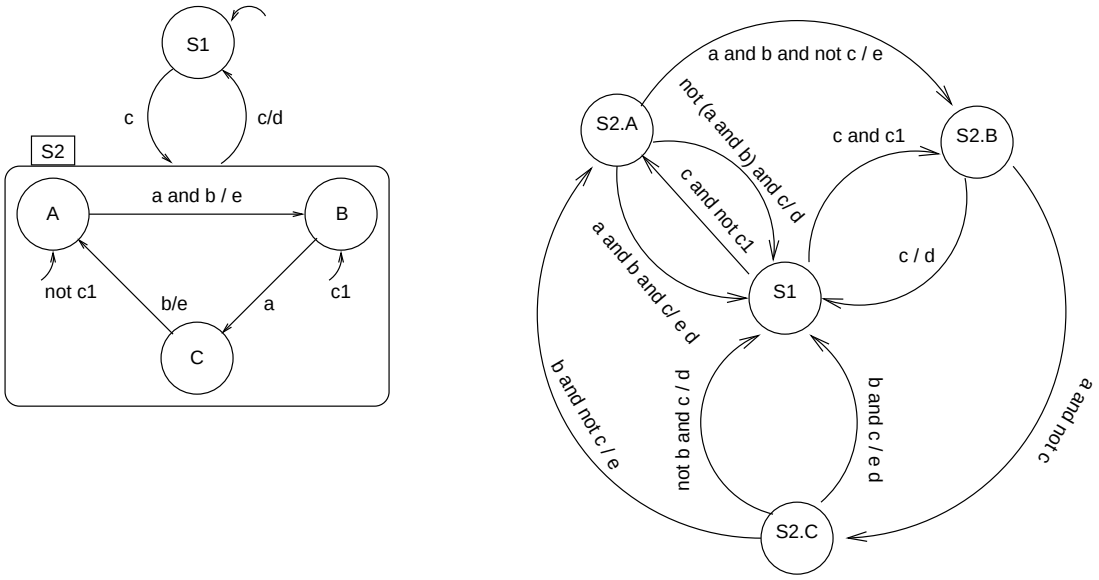


Figure 4: An example of hierarchical composition (left), with the resulting automaton (right).

Let $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{\text{init}}, \mathcal{C}_{\text{init}}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ be an automaton, let $E \subseteq \mathcal{Q}$ be a set of states and let $q \in \mathcal{Q}$ be a state a $\mathcal{A}$. We now define some temporal properties that will be useful in the next section to express the control objectives that will have to ensured on a system modeled as a synchronous automaton.
We say that a state q is *reachable* for $\mathcal{A}$ whenever there exists a trace $t = \{(v_i, q_i, O_i)\}_{i\geq 0} \in Trace(\mathcal{A})$ that traverses q, i.e. $\exists i \geq 0, q_i = q$.

Definition 6 *Let $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{init}, \mathcal{C}_{init}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ be a synchronous automaton and $E \subseteq \mathcal{Q}$, then*

- *$\mathcal{A}$ satisfies the* Reachability of E *whenever there exists $q \in E$ that is reachable for $\mathcal{A}$.*
- *$\mathcal{A}$ satisfies the* Strong Reachability of E*, whenever for all reachable states q, there exists a trajectory initialized in q that reaches E.*
- *$\mathcal{A}$ satisfies the* Invariance of E *whenever $\forall t \in Trace(\mathcal{A})$, s.t. $t = \{(v_i, q_i, O_i)\}_{i\geq 0}$, $q_i \in E$, $\forall i \geq 0$, i.e. $\forall q_{init} \in \mathcal{Q}_{init}$, whatever the behavior of $\mathcal{A}$ initialized in q_{init}, all the traversed states belong to E.*
- *$\mathcal{A}$ satisfies the* Potential Invariance of E*, whenever for all reachable states $q \in E$,*

$$\forall t \in \textit{Sub-trace}(\mathcal{A}, q), \quad s.t.\ t = \{(v_i, q_i, O_i)\}_{i\geq 0}, \quad q_i \in E,\ \forall i \geq 0\ .$$

  i.e. once $\mathcal{A}$ reached one of the states of E, then it will remain inside forever. Note that the potential invariance of E is less restrictive than the invariance of E in the sense that it does not require the system to be initially in E.
- *$\mathcal{A}$ satisfies the* Persistence *of E, whenever A satisfies both the Strong Reachability of E and the* Potential Invariance of E

One can also consider more intricate safety properties that can be modeled by means of particular synchronous automata, called *Observer*

Definition 7 *An* observer *for a synchronous automaton $\mathcal{A}$ is a deterministic and reactive synchronous automaton $\omega = (\mathcal{Q}_\omega, \mathcal{Q}_{init\omega}, \mathcal{C}_{init\omega}, \mathcal{V}_\omega, \mathcal{O}_\omega, \mathcal{T}_\omega)$, such that*

- $\mathcal{V}_\omega = \mathcal{V}_A \cup \mathcal{O}_A$, $\mathcal{O}_\omega = \emptyset$
- $Error \in \mathcal{Q}_\omega$ *is a trap state such that* $(Error, true, Error) \in \mathcal{T}_\omega$.

An observer expresses the negation of a safety property of a system, hence the state *Error* can be seen as a "bad" location which is entered when the system violates the property. The verification consists in checking whether the composition of the two synchronous automata (the plant and the observer) reaches the state *Error* on the observer's side.

2.2 Controller Synthesis

As usual in the controller synthesis setting, the events labeling the transitions can be partitioned into those that can not be controlled (e.g. inputs received from sensors, failures) and those of which the value can be determined or constrained, typically by a discrete controller (typically the starting of some task). The former are called *uncontrollable*, and the latter *controllable*. Note that, in the reactive automaton framework, events can occur simultaneously. Hence a transition between two consecutive states can be labeled by a vector or conjunction of events (some controllable, some others uncontrollable). This constitutes one of the main differences with [27]. In our case, transitions are partially controllable, whereas in the Ramadge & Wonham formulation, the events and transitions are either controllable or uncontrollable. See also [1, 29, 33] for works related to the control of synchronous automata.

2.2.1 Controllers

Let $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{\text{init}}, \mathcal{C}_{\text{init}}, \mathcal{V}, \mathcal{O}, \mathcal{T})$ be an automaton. We partition the set $\mathcal{V}$ of variables into the set of *uncontrollable* variables $\mathcal{V}_U$ and the set of *controllable* variables, $\mathcal{V}_C$. We note the set of valuations on controllable (resp. uncontrollable) variables $\mathcal{V}al(\mathcal{V}_C)$ (resp. $\mathcal{V}al(\mathcal{V}_U)$). Each valuation of the variables (v^u, v^c) contains an uncontrollable component v^u and a controllable one v^c. We have no direct influence on the v^u part which depends only on the state q, but we can observe it. On the other hand, we have full control over v^c and we can choose any value of v^c provided it is admissible.

The behavior of a synchronous automaton $\mathcal{A}$ can be controlled (and therefore restricted) by first restricting initial states and by choosing suitable values for the controllable variables $v_0^c, v_1^c, ..., v_i^c, ...$ to restrict the evolution. Various strategies can be chosen to determine the value of the controls v_i^c's.

We will here consider control policies where the value of the controllable variables v^c is statically computed from the current state and the valuation of the uncontrollable variables. Such a controller is called a *static controller*. It is given by:

Definition 8 *A* controller *$\mathcal{C}_\mathcal{A}$ of $\mathcal{A}$ is given by a pair $(\mathcal{C}_i, \mathcal{C})$, where*

- $\mathcal{C}_i \subseteq \mathcal{Q}_{init}$ *is the restricted set of initial states.*
- $\mathcal{C}$ *is a function* $\mathcal{C} : \mathcal{Q} \times \mathcal{V}al(\mathcal{V}_U) \longrightarrow \mathcal{B}ool(\mathcal{V}_C)$. •

For a given state $q \in \mathcal{Q}$ and a given valuation of the uncontrollable variables $v_u \in \mathcal{V}al(\mathcal{V}_U)$, $\mathcal{C}(s, v_u)$ is a boolean predicate over the variables $\mathcal{V}_C$, such that $\mathcal{C}(q, v_u)(v_c) = true$ means that the controller allows the controllable variables to valuate to v_c. In other words, given the current state

of the system and the value of the uncontrollable events, the goal of a controller is to choose among the values of the controllable variables, the ones that will make the system evolve into the states satisfying the properties from which the controller has been built.

In the framework depicted in Figure 5, the control strategy is the following: given a state q and a set of uncontrollable events that occurs, the set of controllable events that may occur is given by the controller according to the restrictions on transitions computed during the synthesis phase.

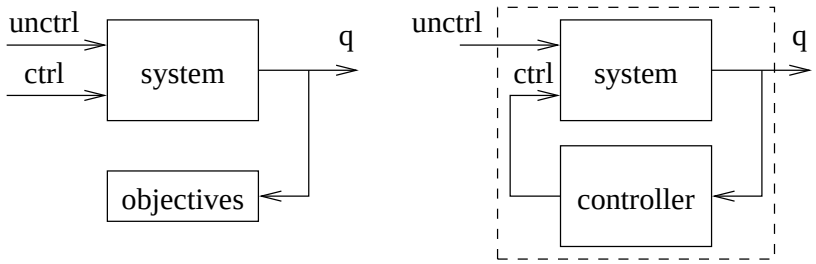


Figure 5: Discrete control synthesis: from uncontrolled system (left) to closed-loop (right).

The automaton $\mathcal{A}$ controlled by the controller $\mathcal{C}_\mathcal{A}$ is another automaton $(\mathcal{Q}^\mathcal{C}, {\mathcal{Q}_{\text{init}}}^\mathcal{C}, \mathcal{V}, calO, \mathcal{T}^\mathcal{C})$, noted $(\mathcal{A}, \mathcal{C}_\mathcal{A})$, such that $\mathcal{Q}^\mathcal{C} = \mathcal{Q}$, ${\mathcal{Q}_{\text{init}}}^\mathcal{C} = \mathcal{C}_i$, and $\mathcal{T}^\mathcal{C} \subseteq \mathcal{Q} \times Bool(v) \times \mathcal{Q}$ is such that

$$t = (q, B(\mathcal{V}), O, q') \in \mathcal{T} \Leftrightarrow (q, B(\mathcal{V}) \wedge \mathcal{C}(q, \mathcal{V}_u)(\mathcal{V}_c), O, q') \in \mathcal{T}^\mathcal{C}$$

And no other transitions are allowed.

However, not every controller $\mathcal{C}_\mathcal{A} = (\mathcal{C}_i, Contr)$ is acceptable : first, the restricted system $(\mathcal{A}, \mathcal{C}_\mathcal{A})$ must have initial states ; thus, $\mathcal{C}_i$ must not be empty. Furthermore, whenever an uncontrollable valuation $\mathcal{V}^u$ of $\mathcal{V}_U$ is admissible in $\mathcal{A}$, then it must be admissible in $(\mathcal{A}, \mathcal{C}_\mathcal{A})$. Hence the following definition of an *acceptable controller*:

Definition 9 *An acceptable controller for $\mathcal{A}$ is a controller satisfying:*

1. *$\mathcal{C}_i \neq \emptyset$, and*
2. *for each reachable state q in $(\mathcal{A}, \mathcal{C}_\mathcal{A})$, any possible valuation v^u of $\mathcal{V}_U$ admissible in q for $\mathcal{A}$ is also admissible in $(\mathcal{A}, \mathcal{C}_\mathcal{A})$.* •

There can be several controllers satisfying the control objectives; actually, sometimes forbidding any move is a control which avoids the states not satisfying the property, but this is less than satisfactory w.r.t. the activity of the control system. The notion of maximally permissive controller captures that we have the controller which insures the properties satisfaction while keeping the greatest subset of behaviors of the original, uncontrolled, system.

Using symbolic methods [21] (based on BDD techniques, avoiding state space enumeration), we can compute controllers $\mathcal{C}_\mathcal{A}$ which ensure

- the *(Potential) invariance* of a set of states,
- the *(Strong) reachability* of a set of states from the initial states of the system
- the *persistence* of a set of states [21].
- etc.

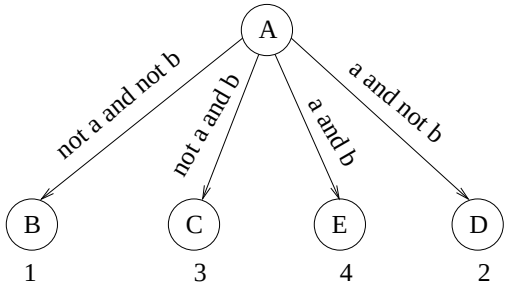


Figure 6: An example of automaton with costs.

For more details on the way controllers are synthesized and algorithms, the reader is referred to [23].

If the property is expressed by means of an observer ω, with a sink state $Error$, then it is sufficient to perform the synchronous product between the automaton $\mathcal{A}$ modeling the plant and the observer and to compute a controller that avoids states of the form $(q, Error)$ to be reachable in $\mathcal{A} \parallel \omega$.

2.3 Optimal Controller Synthesis

On the bases of the notions introduced on the previous section, we now introduce the optimal discrete control synthesis problem that is based on the notion of cost function.

2.3.1 Automata Extension

In order to take into account the notion of levels of e.g., quality, time or energy consumption, in the control objectives, let us first extend the synchronous automata definition:

Formally, an automaton with costs $(\mathcal{A}, Costs)$ consists of an automaton $\mathcal{A} = (\mathcal{Q}, \mathcal{Q}_{\text{init}}, \mathcal{C}_{\text{init}}, \mathcal{V}, \mathcal{T})$ and of a *state cost function* $Costs$, defined as follows: $Costs : \mathcal{Q} \longrightarrow I\!N$, where the set $I\!N$ represents the set of naturals.

Example 4 *Figure 6 illustrates an automaton with costs.*

Composition Operators The operators of composition are naturally extended to automata with costs. The extensions need to define how costs are composed on states: the operators composing costs on state do not require any hypothesis *a priori*, except that their result domain has to be $I\!N$. The most common operators are among $+$, $*$, max... Let $(\mathcal{A}, Costs)$ be a cost automaton We note OP the composition operator on the state cost. Note that it is possible to attach a cost vector to the states. In this case, the operators composing costs on state may be different according to the meaning of the cost.

Synchronous Product. Let $(\mathcal{A}_1, Costs_1)$ and $(\mathcal{A}_2, Costs_2)$ be two cost automata. Then, the *synchronous product of* $(\mathcal{A}_1, Costs_1)$ *and* $(\mathcal{A}_2, Costs_2)$ is the cost automaton $(\mathcal{A}_1, Costs_1) \parallel (\mathcal{A}_2, Costs_2) = (\mathcal{A} \parallel \mathcal{A}', Costs)$, where

$$\forall \langle q_1, q_2 \rangle \in Q_{\mathcal{A}\parallel\mathcal{A}'},\ Costs(\langle q_1, q_2 \rangle) = Costs_1(q_1)\ OP\ Costs_2(q_2)$$

Hierarchical composition. Let $(\mathcal{A}_b, Costs_b)$ and $(\mathcal{A}_r, Costs_r)$ be two cost automata. Let $q_r \in \mathcal{Q}_b$ be a state of $\mathcal{A}_b$, to be refined by $\mathcal{A}_r$. The result of this refinement is the cost automaton

$$(\mathcal{A}_b, Costs_b) \triangleright (q_r, (\mathcal{A}_r, Costs_r)) = (\mathcal{A}_b \triangleright (q_r, \mathcal{A}_r), Costs)$$

where $\forall q, Costs(q) = Costs_b(q)$ and $\forall q_r \,.\, q,\ Costs(q_r \,.\, q) = Costs_b(q_r) OP_r Costs_r(q)$, where OP_r is a state cost function making the link between the high and low level of the hierarchical cost automaton.

Bounding Properties Let us now go through some properties that may be checked on a cost automaton. They will be used as the basis to express control objectives in the next section.

Local Bounding Property This property ensures that every reachable state (from the initial ones) has bounded costs. Let $(\mathcal{A}, Costs)$ be a cost automaton, let $Low, Up \in \mathbb{N}$. The cost automaton $(\mathcal{A}, Costs)$ is locally bounded iff $\forall t = \{(v_i, q_i)\}_i \in Trace(\mathcal{A}), \forall i > 0,\ Low \leq Costs(q_i) \leq Up$.
This property can be easily extended to finite traces as follows:

Bounding Property on Traces The cost of a trace $t = \{(v_i, q_i)\}_i \in Sub\text{-}trace(\mathcal{A}, q)$ within $K \geq 0$ steps is defined by $Costs(t, K, q) = \sum_{i=1..K} Costs(q_i)$. Let Low, Up be integers. The cost bound on traces within K steps is verified iff $\forall q, \forall t \in Sub\text{-}trace(\mathcal{A}),\ Low \leq Costs(t, K, q) \leq Up$.

2.3.2 Optimal Controller Synthesis Problem

Ensuring bounding properties Let $(\mathcal{A}, Costs)$ be a cost automaton and let $\mathcal{V} = \mathcal{V}_U \cup \mathcal{V}_C$ be the partition of variables of $\mathcal{A}$ into uncontrollable and controllable. Let $Low, Up \in \mathbb{N}$ be integers. The controller synthesis problem for local bounding (resp. bounding on traces) consists in finding a controller $\mathcal{C}$ of $\mathcal{A}$ such that the controlled system of $\mathcal{A}$ by $\mathcal{C}$ is locally bounded (resp. bounded on traces) by Low, Up. The above properties are still logical properties expressed on costs. Indeed, to ensure this property it is sufficient to first compute the set of states $E = \{q \in \mathcal{Q} \mid Low \leq Costs(q) \leq Up\}$ and to make it invariant using the methodology describes in Section 2.2.1.

Example 5 *For example, considering the automaton with costs described in figure 6, if one wants to remain in the states $E = \{q \in \mathcal{Q} \mid 2 \leq Costs(q) \leq 3\} = \{A, C, D\}$, then assuming the system is in state A, then if $a = true$ the controller will force $b = false$ (to avoid E) and if $a = false$ the controller will force $b = true$ (to avoid B).*

Optimization properties are also defined using costs. The idea is to make the system evolve into state with the highest (resp. lowest) cost w.r.t. its current position and the valuation of the uncontrollable variables $\mathcal{V}_U$. Intuitively speaking, the cost function is used to express priority between the different states that a system can reach in one transition.
Formally, if it is maximization under consideration, let $(\mathcal{A}, Costs)$ be a cost automaton, given a state q and an admissible uncontrollable valuation v^u, then the valuation v_1^c is said to be better compared to the valuation v_2^c whenever, the states q_1 and q_2 s.t. $(q, B, O, q_1) \in \mathcal{T} \wedge B((v^u, v_1^c)) = true$ (resp. for q_2) are such that $Costs(q_1) \geq Costs(q_2)$.
In other words, the controller has to choose, for a pair (q, v^u), a control compatible with v^u in q, that allows the system to evolve into one of the states that has a maximal cost. Hence, if the system is in state q and the valuation of $\mathcal{V}_u$ is received, the set of suitable valuations of $\mathcal{V}_c$ is

$$\begin{aligned}\mathcal{I}_{max}(q, v^u) = \{v^c \in val(\mathcal{V}_C) \ / \exists (q, B, O, q') \in \mathcal{T},\ \text{s.t. } B((v^u, v^c)) = true \wedge \\ \forall v^{c\prime} \text{ s.t. } (q, B', O', q'') \in \mathcal{T},\ \text{s.t. } B'((v^u, v^{c\prime})) = true,\ Costs(q') \geq Costs(q'')\}\end{aligned}$$

Based on this policy, one can easily derive a controller $\mathcal{C} = (\mathcal{C}_{init}, \mathcal{C}_{\mathcal{A}})$, such that $\mathcal{C}_{init} = \mathcal{Q}_{init}$ and $\mathcal{C}_{\mathcal{A}}$ is such that $\forall q \in \mathcal{Q},\ \forall v^u$ admissible in $q\ \mathcal{C}_{\mathcal{A}}(q, v^u)(v^c) = true \Leftrightarrow v^c \in \mathcal{I}_{max}(q, v^u)$.

The minimization property may be similarly defined by replacing "max" with "min" in the above equations.

Example 6 *Coming, back to Example 4, the result of the above computation will give access to an automaton such that B and D are forbidden by control. Indeed, if $a = false$ then depending on the value of b the system will evolve either in B or C. However $Costs(C) > Costs(B)$, therefore, the controller will force $b = true$ to make the system evolve in C. For the same reason, if $a = true$, then , the controller will force $b = true$ to make the system evolve in E.*

Cost function composition. When considering two criteria, e.g., quality and energy, you may want to first maximize quality w.r.t. the cost function C_1, and then minimize energy w.r.t. C_2. Given a current state q and a valuation of the uncontrollable variables v^u, for the first objective, you follow the previous methodology, and obtain a set $\mathcal{I}_{max}(q, v^u)$ that are solutions of $\mathcal{C}^1_{\mathcal{A}}(q, v^u)(\mathcal{V}_c)$. Then, the set of suitable valuations for the controllable variables are given by:

$$\mathcal{I} = \{v^c \in \mathcal{I}_{max}(q, v^u) \ / \exists (q, B, O, q') \in \mathcal{T}, \text{ s.t. } B((v^u, v^c)) = true \wedge$$
$$\forall v^{c\prime} \in \mathcal{I}_{max}(q, v^u) \text{ s.t. } (q, B', O', q'') \in \mathcal{T}, \text{ with } B'((v^u, v^{c\prime})) = true, C_2(q') \leq C_1(q'')\}$$

i.e. among the remaining valuations of the controllable variables, we only keep the ones that minimize C_2.

The same kind of techniques can be used if one wants to mix event and state costs. These optimal synthesis functionalities are also implemented efficiently in SIGALI, and can be used for experiment, as outlined in the next section.

2.4 Tools implementing the models and synthesis

The current implementation of the method, which has been used for the example, relies on the chain of Figure 7. Centrally, we use SIGALI [21], which is a tool that performs model-checking, controller synthesis for logical goals, and optimal controller synthesis.

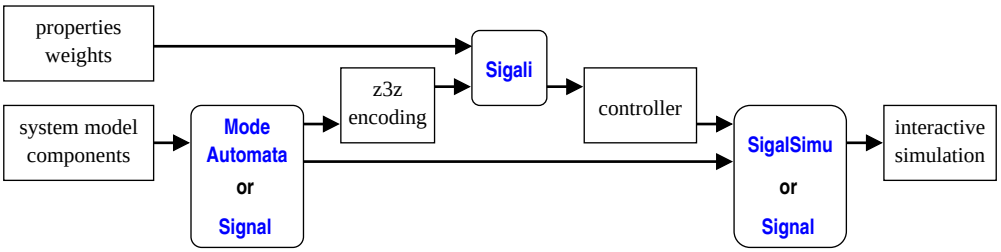


Figure 7: Implementation of the approach: the tools involved.

The components behaviors, task activations schemes and properties observers, can be describes using a synchronous formalism. The equational language SIGNAL is the synchronous language originally connected with the synthesis tool SIGALI [21]. Another such formalism is Mode Automata [19], for which the tool MATOU provides for compilation and has been adapted to generate the `z3z` format, the input format of *Sigali*. The global properties and the weights are expressed into `z3z` by the means of SIGALI macros.

The result of the synthesis in SIGALI is a controller, in the form of a logical relation, which can be interpreted by a resolver module: for a given state and uncontrollable input, the constraints on

controllable signals are solved, for example in an incremental, interactive way following the manual valuation of signals.

The resolver can be coupled with the original specification of the uncontrolled system, using either SIGNAL, or the tool *SigalSimu* in the case of Mode Automata.

A perspective is to transform the interpretation chain into a *compilation* chain, producing the controlled system as an explicit automata that can then be compiled into C code.

3 Modeling tasks for a safe handling

In this section, we present in Section 3.1, the class of applications which we are going to model: multi-task control systems; in Section 3.2, patterns for modelling the control of such tasks, and of a multi-task system; in Section 3.3 the logical properties which can be considered as objectives for the global system control; and in Section 3.4 the objectives of cost and quality of service. In the presentation, we will use the graphical syntax for the transition systems introduced before.

3.1 Informal motivation of the model

Multiple cyclic tasks in parallel. We consider reactive systems implementing a set of cyclic control tasks, classically by executing, in an endless loop:

- reading of inputs,
- computation of the control and update of internal state variables,
- emission of the outputs

A more elaborate design can be considered, when the computation is subdivided into sub-tasks with possibly different periods and synchronizations, but the global cyclic scheme remains the one adopted in reactive systems, in StateCharts [13], or in the synchronous approach [11, 12].

A complex system can be constructed in terms of tasks, each with an idle state, from which they can be started, going to an active state to which the computation is associated. Other significant control states may feature an initialization state, where some internal state variables are computed (e.g., filters) for a number of cycles, or exception handling state, where computation is different to the nominal active state for a number of cycle. A variety of such control patterns can be envisaged [5, 20].

When having a set of such tasks in parallel, each global reaction or cycle will see the performance of the computations associated with each of the active tasks. Hence, when implemented on a single processor architecture, the computations are executed in sequence, and the duration of a reaction is the sum of individual durations for active task cycles. Indeed, the so-called "zero-time" hypothesis of synchronous languages is a metaphor of the fact that interactions inside a cycle can be compiled away, and hence costless; actual implementation has a worst-case execution time (WCET) to be measured w.r.t. the environment dynamics.

Modes and their characteristics. If one considers tasks with a variety of possible active modes, such as proposed in the Mode Automata [19, 20], then one can consider that these modes are differentiated by some characteristics, such as, e.g.:

- time cost, i.e. the duration taken by one cycle of computation (e.g., each mode has a particular WCET which takes part in the global reaction);

- quality (e.g., precision of numerical computation);
- the use of bandwidth in heavily communicating systems (you may go into a mode using more time but less bandwidth, e.g. by applying compression algorithms);
- energy consumption, in portable devices or autonomous systems.

In the present proposal cost and quality are closely related in the sense that the higher the quality delivered by the mode implementing the functionality, the higher the corresponding cost. When considering multiple tasks running in parallel (meaning: sharing processor time within each cycle) time costs naturally combine additively; for quality, we will also consider additive combination in this paper: although it is a bit simplistic, it facilitates explanation of the approach. However this does not in principle exclude other interpretations.

Switching between modes, and its safe handling. Once a set of tasks with modes is defined, one has to consider the switching between them. A task is initially idle, and a request is awaited for; it can come from an application written on top of the task set, from a system end-user command, or from other tasks or sensors. Within a multi-task environment, starting a new task can occur in several ways: when enough computing resource is available, it can be started right away; when some computing resource has to be made available, it can be achieved by lowering quality (hence time cost) of other already active tasks, through mode switching, while keeping global quality maximal; when no sufficient resource can be released, then we have to go to a waiting state.

This kind of control, managing mode switches according to criteria of time cost and quality level, is what we want to obtain automatically, through discrete control synthesis. For this, we will now propose automata modeling patterns for classical control of the activity of tasks, providing for some control on their mode switching.

3.2 Task Activation Schemes

3.2.1 Simple task activation schemes

Standard tasks, with application request. As shown in Fig. 8, initially, a task is `Idle`. It goes from `Idle` to `Act` when there is a request (through the uncontrollable event `r`) and the controller accepts it (through the controllable event `go`), i.e. the control constraints allow it. With the intention of "installing" controllability in the model, a `Wait` state has been incorporated to enable the recording of a request when the activity of another task prevents the controller from starting it. The controller may choose to make it active once the conditions are favorable. Termination of a task is signaled by the event `stop`. Under this model, only the event `go` is assumed to be *controllable*; the others are *uncontrollable*.

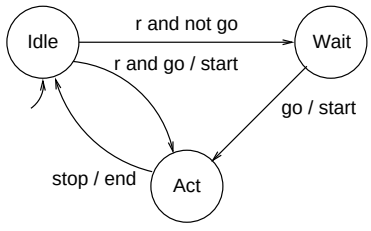


Figure 8: The discrete model of a *standard* task.

This model features emitted events, signaling actual `start` and `end` of the activity, which will be used by the observers as described in Section 3.3.

Default tasks, fully controllable. Robots or control systems often require to be always under control, even for rest configurations, because of gravity or other external forces. This motivates the introduction of a pattern for default tasks, shown in Fig. 9.

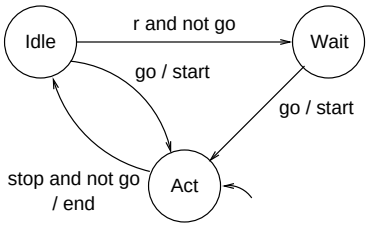


Figure 9: The discrete model of a *default* task.

It is similar to the standard task except that it is not necessary to have a request in order for a default task to become active. However, if a request is made by the environment (e.g., the operator or the user) without the permission of the controller (i.e., `not go`), then the task evolves into the `Wait` state, as before. Also, when the event `stop` is received, the controller decides upon the termination of the task, by triggering `not go`. This task can be initially in the active state, meaning that it is activated when the system is switched on, before any request for a task is to be considered.

The meaning of the `stop` event is that it is received at each instant where the task is ready to terminate. For example, in the case of control tasks, it can correspond to the fact that a control law has reached a point where it is close enough to the goal, so it can go on staying close to it, or terminate in order for the control to proceed with another task in sequence. Otherwise, the task remains active. It means that it is the controller which decides whether a task has to be stopped or not.

The default task gives the controller more power and restriction possibilities.

Sustainable task. Another interesting pattern is the one illustrated in Figure 10, where the possible stopping of the activity (signaled by the event `stop`) can be memorized, but the activity sustained. The behavior is quite symmetrical to the memorization and control of requests seen above. A specific state is used to explicitly distinguishing that a stop event has been received once, but that actual stopping and going to the idle state is controlled by the controller, which can choose to sustain the activity if it is required in the waiting for some other task to become active in sequence.

An example is the class of tasks in control systems where a control loop has met its objective (e.g., reaching a position), but no other control task has been scheduled in sequence. Sustaining the current task makes that the control objective is regulated further (e.g., the device is kept at the position reached), and the corresponding actuators a still under control.

Other task patterns. The preceding patterns are simple particular cases. Other possible extensions, illustrated in Figure 11 would be:

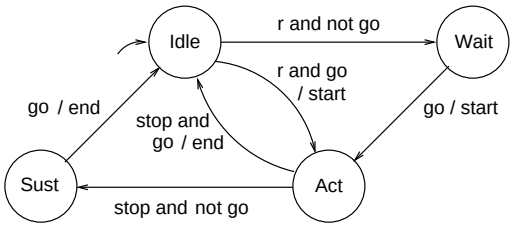


Figure 10: The discrete model of a *sustainable* task.

- to allow the controller to reject requests (i.e., no wait/memorization), with emission of a rejection warning (such a pattern imposes quite strong constraints on a multi-task system, of course);
- the possibility for the environment or application to cancel a request (e.g., in case of abortion or preemption on the requesting process),
- error states (corresponding to fault events, and reparation activities),

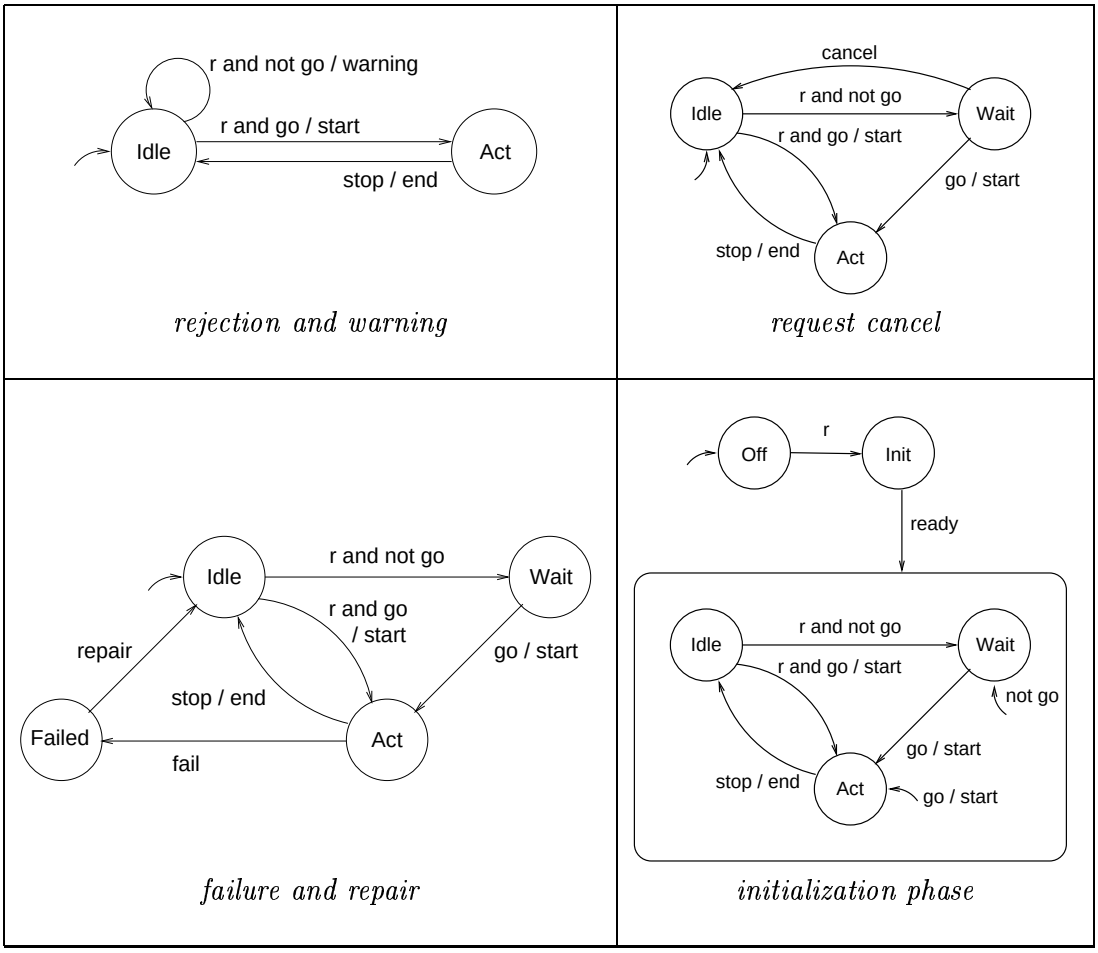


Figure 11: Patterns of tasks.

- the representation of an initialization phase, to be fulfilled before control can be taken over by the task;
- .....

Such variants on the patterns can be inspired by the various approaches to the application of control synthesis, to robotic control systems [24], property-enforcing layers [2], or fault tolerance [10].

3.2.2 Hierarchical task schemes

Hierarchical tasks. Motivations for hierarchical structure in control tasks are that it favors the well-structured design of programs, by hiding details of lower-level parts when considering the transitions of the upper-layers, and that it makes it possible to reuse local controllers by assembling them in different ways. Hierarchical structures in automata-based languages are typically associating a sub-automaton to a state of an automaton, like in StateCharts [13] or Mode Automata [19]. The meaning is that, when the upper-level automaton is in a state, then the sub-automaton reacts to inputs according to its behavior; and when the upper-level automaton makes a transition out of the state, then the sub-automaton is preempted : it is not active any more.

A particular case of hierarchical structure is that of tasks that can be executed according to different modes or versions. The notion of modes and their switching is approached in various ways, some related to operating system mechanisms, or language constructs [28]. Some works address the schedulability analysis of the combination of the task schedulings for each mode and for the transition itself [8, 35]. We focus on the case of cyclic real-time systems in the synchronous approach to reactive systems [11], and adopt the notion of modes as in Mode Automata [19]. We are interested in real-time control systems with multiple tasks, each with multiple modes, which can be versions implementing the same functionality with different levels of quality (e.g., computation approximation), and cost (e.g., computation time, energy, side-effects on the environment). It is complex to control the switching of modes in order to insure properties like bounding cost while maximizing quality (i.e., limiting degradation).

Once a set of tasks with modes is defined, one has to consider the switching between them. A task is initially idle, and a request is awaited for; it can come from an application written on top of the task set, from a system end-user command, or from other tasks or sensors. Within a multi-task environment, starting a new task can occur in several ways: when enough computing resource is available, it can be started right away; when some computing resource has to be made available, it can be achieved by lowering quality (hence time cost) of other already active tasks, through mode switching, while keeping global quality maximal; when no sufficient resource can be released, then we have to go to a waiting state. This kind of control, managing mode switches according to criteria of time cost and quality level, is what we want to obtain automatically, through discrete control synthesis.

A multi-mode task. Fig. 12 gives the hierarchical automaton for a task, say T_i, for the case with three modes. The pattern formalizes the aspects described above. We have a first level with states: $\texttt{Idle}_i$ (or $\texttt{I}_i$), where the task is deactivated; $\texttt{Wait}_i$ (or $\texttt{W}_i$), where it has been requested, by an event $\texttt{req}_i$, but is not launched yet, because of lack of authorization $\texttt{Go}_i$ by the controller (due to constraints with the environment or other tasks); $\texttt{Act}_i$ (or $\texttt{A}_i$), where it has been requested, and authorized through $\texttt{Go}_i$, and hence is active.

Within the active state $\texttt{Act}_i$, several modes can be defined. Each task T_i has m_i modes $M_{ij}, 1 \leq j \leq m_i$. For each task T_i, transitions between modes are labeled by conditions c_{k_i}, managed by the

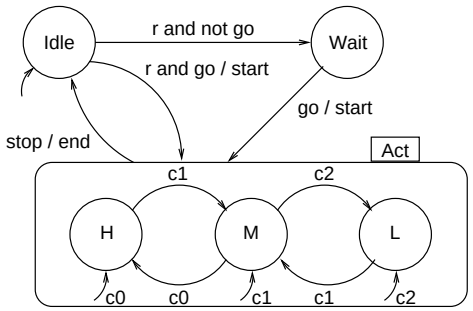


Figure 12: Task pattern with multiple modes.

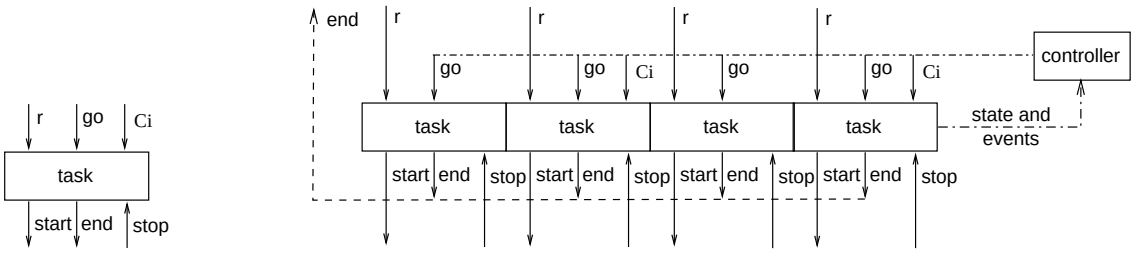
controller in order to authorize the switch or not. The mode which becomes active when entering $\texttt{Act}_i$ is chosen according to the one of these conditions which is true. The choice of the mode when starting is determined by the control values.

In the case of three modes, illustrated in Figure 12, we can think of applications such that we have: $\texttt{H}_i$ (highest-quality and time); $\texttt{M}_i$ (medium); $\texttt{L}_i$ (low). These three modes can be switched between according to the transitions and their conditions c_{k_i}. In particular, you have to go through the $\texttt{M}_i$ mode between $\texttt{H}_i$ and $\texttt{L}_i$. An example is a task computing at each cycle an expression summing three terms [20]: $E = E_1 + E_2 + E_3$, where E_3 and E_2 can be approximated by 0. Each of the modes corresponds to: $\texttt{H}_i$: the full sum, $\texttt{M}_i$: an approximation $E_1 + E_2$, $\texttt{L}_i$: a degraded version E_1. In the following, we will see how valued characteristics can be associated to modes.

3.2.3 Multi-task systems

The tasks that were modeled previously can be assembled into multi-task systems. Their individual interface basically consists of the inputs and outputs illustrated in Figure 13(a).

Tasks server. In order to model a multi-task system, we consider just the parallel composition of the n tasks, as in synchronous languages [11, 19], in accordance with the definition in section 2.1.1. Hence, a global state or configuration S is described by a vector of state values, one for each task, giving the current active mode. Figure 13(b) illustrates such a parallel composition of 4 tasks, the first and third from the left without modes. Each receives its requests and stop events, as well as the controls `go` and possibly `Ci`, from the controller which has access to all state and event information. Each task also sends out its starts and ends to the environment, as control instructions on the side of the tasks computation processes, or as informations towards an upper level.



(a) *multi-mode task interface* (b) *tasks server, with a controller*

Figure 13: Tasks interfaces (with mode control) (a), and tasks server (b).

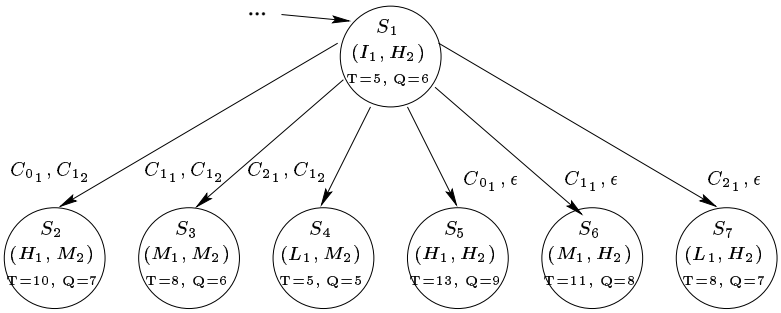


Figure 14: Configurations reachable in one step from S_1, upon reception of ($\texttt{Req}_1$ `and` $\texttt{Go}_1$) and in the absence of $\texttt{Stop}_2$, according to controllables $C_{k_1}, C_{k'_2}$, with weights of cost and quality.

For an example with two instances T_1 and T_2 of the three-mode pattern of Figure 12, we can be in a mode where T_1 is in state $\texttt{Idle}_1$ while T_2 is in state $\texttt{Act}_2$, in mode $\texttt{H}_2$: the configuration can be noted in short $S_1 = (\texttt{I}_1, \texttt{H}_2)$. It will be useful in the following to define the number m of active modes in a configuration S. One can observe that $m \leq n$, as at most one mode is active per task. As illustrated in Figure 14, a global step is taken when one or several mode changes or a task start and/or stop event occurs, going from one global configuration to the next. Such a transition is labeled with a vector of conditions/events of the local transitions, or ϵ on the side where no transition is taken. In the example, when in the configuration S_1, upon the occurrence of ($\texttt{Req}_1$ `and` $\texttt{Go}_1$) and in the absence of $\texttt{Stop}_2$, the configurations shown in Figure 14 can be reached in one step. They are reached through transitions which are labeled by, for T_1: either one of $\texttt{c0}_1$, $\texttt{c1}_1$, $\texttt{c2}_1$, and for T_2: either $\texttt{c1}_2$ or ϵ (empty label, no movement of T_2). E.g., going from S_1 to $S_2 = (\texttt{H}_1, \texttt{M}_2)$ would be through a transition labeled with a combination of the two involved local transitions: ($\texttt{Req}_1$ `and` $\texttt{Go}_1$ `and` $\texttt{c0}_1$), $\texttt{c1}_2$).

Synchronizations and applications. Complete systems involving multiple tasks can be constructed by adding an application program on top of the tasks server. One way of combining tasks is to have a language of "loose" synchronization, where part of the scheduling is expressed by the user, in the form of a script or a scenario, and controller synthesis serves as a completion, computing the part relevant for constraints management. A task-level language can be used to describe sequence,

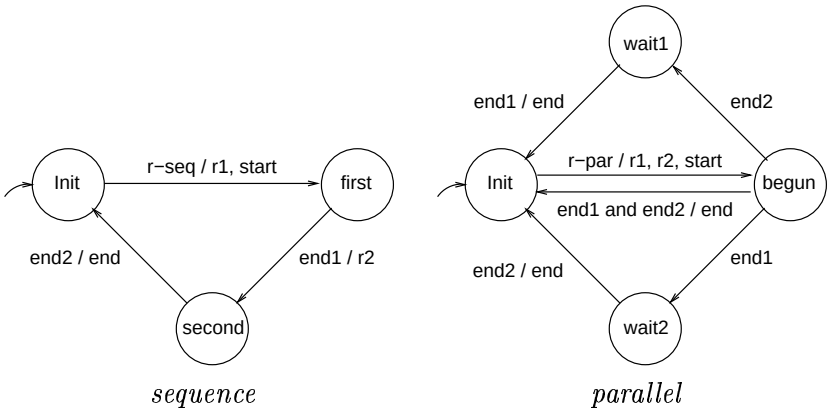


Figure 15: Sequence and parallel of two tasks.

parallelism, reaction to events, resulting is a sequence of requests, themselves controlled, e.g. in an interpreted way, by the automatically synthesized controller, ensuring satisfaction of the appropriate properties. Automata for the sequence and parallel of two tasks are illustrated in Figure 15 where, upon reception, respectively, of `req-seq` and `req-par`, requests are emitted accordingly towards the tasks, the ends of which then being awaited for. Requests to tasks are the local events, and not inputs any more. Such a language might look like domain-specific languages like PILOT [34] or MAESTRO [36]; but the synchronizations would be "looser" in the sense that going in sequence from one task to the next would not result in immediate starting, but rather to emission of a request, and eventual starting of task, possibly after some waiting, according to the controller.

The resulting system is illustrated in Figure 16, where, on top of the server of Figure 13, an application component, when receiving a request from its own environment, emits requests towards the tasks, and reacts on their ends, until reaching its own end. The controller can rely on the states and events from the application, in addition to that of the tasks.

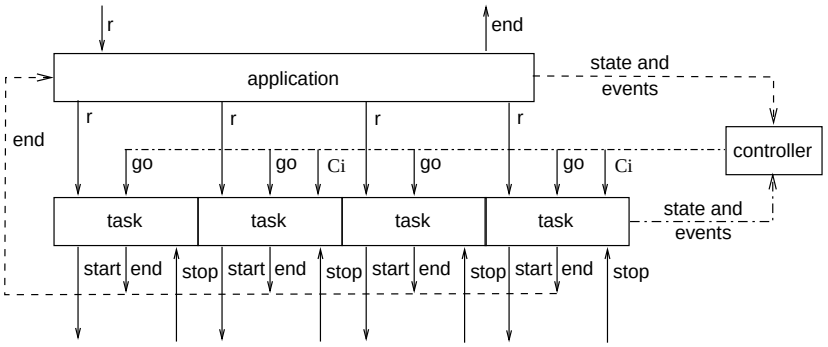


Figure 16: Application on top of a tasks server.

3.3 Associated logical properties

3.3.1 How to specify properties.

Formulation. Properties can concern configurations of the system, defining states that are defined to be consistent or dangerous. They can also concern sequences of events, states or tasks, that can be either forbidden, or required. The objectives for which controllers will be synthesized will consist of making a set of states verifying a property e.g., invariant, or reachable.

Their formulation can involve writing logic formulæ which can be quite technical, especially in the case of temporal logics. There are works contributing to making such specifications more accessible, on the basis of domain-specific knowledge and notion, e.g., in robotics (e.g. proposals for a specialized language of properties in ORCCAD), or also finance [14].

Observers can be used as an alternative. Instead of a predicate on state variables, or a temporal logic formula, they are defined by an automaton, recognizing the sequence, with a terminal state. The global system is the parallel composition of the observer and the pre-existing system. It can be submitted to a control objective of safety, keeping out of the terminal error state. This method has the advantage of making it possible to use the same specification language as for behaviors. As an example, to ensure that the system avoids a certain sequence of task activations, one can define a transition system recognizing that sequence, with a final state designating the error. The reachability of this state is then a simple property that can be used for verification or synthesis.

Generic, architecture-specific, and application-specific properties. In a way similar to the proposal of typical task control schemes, we want to propose property schemes typical to the class of systems under consideration. The fact that we consider a specific domain favors the identification of such schemes, which can be encoded once and for all, and used by engineers without the need to re-formulate them, hence avoiding a risk of misinterpretation and discrepancy with the intended specification. Of course, the possibility remains to encounter the need for more particular properties, specific either to the considered architecture (resources, sensors and actuators, communication network, ...), or even to the application executed on this platform.

We see a gradation beginning, at the most general level, with generic, architecture-independent properties. These properties should be verified, or controlled for, for any particular set of tasks and resources. An example for control systems is that every actuator should always be under control, and that all tasks controlling an actuator should be exclusive. Another example is that access to other exclusive (single-user) resource should be controlled appropriately.

At a more particular level, we see architecture-specific, application-independent properties. They concern aspects that have to be satisfied for all uses of an architecture, i.e., whatever the application executed on the platform, and the sequence of requests and inputs. At this level, one can state that some constraints hold between some particular tasks, whatever the application. An example is tasks of a chemical plant, not using the same pipes or tanks, but required to be exclusive because of potential reaction through fumes. Another example is the interdiction of sequences or combinations of tasks involving manual interaction, because of safety regulations w.r.t. to the level of attention required.

Finally, application-specific constraints can always be part of a design. These properties, having to be determined for each application, are susceptible of being characterized in advance, in a generic way. Therefore, they might involve all the expressiveness of a temporal logic.

3.3.2 Properties on states

Generic objectives on any set of tasks. We have identified a few properties and objectives that can be defined on the task patterns proposed above. They are domain-specific in that they concern a particular abstraction of control tasks, and that the constraints are related to requirements of control systems. At the same time, they are generic w.r.t. that domain, meaning that they are relevant to a wide range of applications, and that they can be significantly used for any instantiation of the given patterns. They can also be systematically derived for robotic missions built from task-level components as we introduced. In that sense, we have a framework where models as well as objectives can be automatically compiled from high-level specifications in a domain-specific language.

They are based on the notion that the system to be controlled is composed of a set of devices or resources (e.g. actuators of a robot) $d \in \mathcal{D}$. The system has a set of tasks $t \in \mathcal{T}$. These tasks are instantiations of the patterns described previously, i.e., for each of them we have the states $idle_t$, $wait_t$, act_t, and we have a predicate active(t) telling us whether or not a task is active, i.e., for a standard or default task, in state Act, or for a sustainable task, in either state Act or Sust. Their composition, together with observer automata in a set $\mathcal{O}$, defines a global automaton.

A function u defines, for a task t, the set $u(t)$ of devices (or resources) used by t: $u : \mathcal{T} \longrightarrow 2^{\mathcal{D}}$. Reciprocally, another function a defines, for a device d, the set of tasks that can make access to d: $a : \mathcal{D} \longrightarrow 2^{\mathcal{T}}$.

Unicity of control for an actuator. A basic property of such a system is that there always is, for each actuator, at least one control law controlling a given actuator (otherwise the actuator is

not under control, and may e.g., fall down), and at most one (otherwise the actuator receives possibly contradictory commands). This can be written as an invariant to be satisfied by all behaviors:

$$\forall d \in \mathcal{D}, \exists! t \in \mathcal{T} \; . \; d \in u(t) \wedge \text{active}(t)$$

In order to distinguish between resources requiring to be under control, and others accepting at most one user, we will decompose this property.

For synthesis purposes, invariance objectives can be expressed on the state information of the control. For this, we define a Boolean condition in terms of the model state variables (i.e., an observer of the situation), and then define the objective as a simple expression, like: achieving invariance of the truth of the value of that Boolean.

Existence at all instants of a control law can be described as:

$$\bigwedge_{d \in \mathcal{D}} \left(\bigvee_{t \in a(d)} \text{active}(t) \right)$$

The objective is to *make this invariantly true.*

Exclusivity of resource access, more specifically of control laws on the same actuator, is another basic property to be maintained in a control system, i.e., at most one control law at a time can be controlling a given actuator. It can be achieved in a way quite close to the previous one: it is a property on states, where the situation *to be avoided* is that for any actuator there is more than one active task, in other words:

$$\bigwedge_{d \in \mathcal{D}} \left(\bigvee_{(t,t') \in a(d)^2, t \neq t'} (\text{active}(t) \wedge \text{active}(t')) \right)$$

The objective is to *make it invariantly false.*

Architecture-specific properties.

Exclusivity between two tasks or modes is a typical state property; it can be related to a common resource as seen above, or to more external issues, specific to the system under consideration. For example, side-effects of different tasks (in terms of temperature, or dispersion of chemicals in the ambient, ...), or safety-related considerations (the fact that it can be possible but dangerous to perform two particular tasks at the same time) can be known to make them incompatible.

This can be captured by a simple expression on the corresponding states; for an exclusivity between task t_i and mode m_{jk}:

$$\text{active}(t_i) \wedge \text{active}(m_{jk})$$

The objective is to *make it invariantly false.*

Reachability of a rest configuration. Some systems may require a procedure to be stopped, more elaborate than simply switching them off. Or similarly, they can have a defined rest position or configuration, into which they are considered safe, and that configuration should always be reachable. Such a configuration can be defined w.r.t. some of the resources and devices in presence; some of them may have to be under the control of a task, which should then be active, the others being idle.

For such a rest configuration defined by an expression on state variables, hence defining a subset of states, the objective is to *make it reachable from all the reachable state space.*

Another use of reachability is to verify that, if a controller was found w.r.t. invariance constraints, it is not so conservative that all tasks are accepting requests only to to be blocked in a waiting state. This can be excluded by defining a state or set of states where the functionality of the system is fulfilled (e.g., a terminal state), and requiring it to be always reachable. An example is in fault-tolerant systems, where for all faults (within the fault model considered), the functionality can still be fulfilled [10].

3.3.3 Task sequences and transitory modes.

Until now properties considered are static in the sense that they concern situations rather than series of transitions. More dynamical properties are related to allowed task sequences and requested transitory modes. As was said before, this can involve defining observers recognizing these sequences, and distinguishing the terminal state (or set of them), in order for it to be used in objectives of reachability or invariance.

Related to this point, an interesting extension would be to characterize objectives about waiting tasks and defining policies for their order of release, which would be implemented by the synthesized controller.

Avoiding t_3 between t_1 and t_2 is an example of property. More precisely, we want to have, between the ending of t_1 and the next activation of t_2, no activation of t_3. Cases with simultaneity are acceptable. For this, an observer can be proposed as in Figure 17.

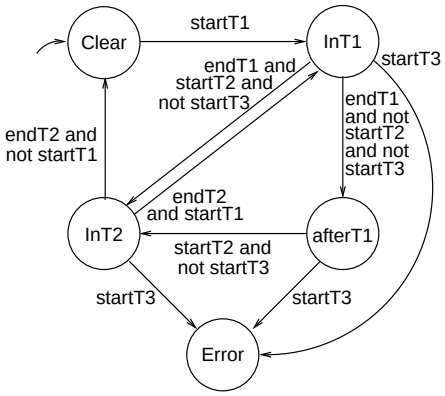


Figure 17: An observer for an activation of t_3 between t_1 and t_2.

This automaton, initially in state `Clear`, observes the start and end events emitted by the tasks; so we have the following transitions. An occurrence of the starting of task t_1 causes a transition to state `InT1`.

There, in case t_3 is started, a transition goes to `Error`. Otherwise, upon the task deactivation `endT1`: in the absence of `startT2` the transition goes to `afterT1`; in the simultaneous presence of `startT2` the transition goes directly to `InT2`.

From `afterT1`, if t_3 starts, we go to `Error`, otherwise, when t_2 starts, we go to `InT2`.

From `InT2`, an occurrence of `startT3` before t_2's end brings to `Error`, otherwise `endT2` brings back to the initial state (where t_3's activity may be observed) or, if we have simultaneously `startT1`, to `InT1` directly.

The desired property is that a sequence reaching `Error` can not be followed. The way to achieve that is to synthesize a controller for the objective: *making invariant the value false for the state variable* `Error`.

Imposing t_2 between t_1 and t_3 can then be seen as avoiding t_3 between t_1 and t_2. In particular, it can be useful in order to have automatically a transitory mode through a default task t_2 between two tasks t_1 and t_3 on the same actuator. This corresponds to the existence of control laws which needs to be separated by a special mode handling the transitory situation between the two (e.g., velocity control followed by position control, or cooling between two tasks soliciting a device which heats up).

3.4 Weights, costs and quality of service

Until now, only strictly logical aspects of the system behaviors and constraints have been considered. We will now assign quantitative characteristics to states and modes, in order to encompass some aspects relating to resource usage and sharing, characterization of quality of the actions performed, and the like. They are typically specified locally to these states, and we define how to deduce them for tasks and applications. They can be used for the synthesis of controllers for which the objective is to keep bounded, minimize or maximize the global weights.

3.4.1 Modes, tasks and applications characteristics

We define a cost function representing quality C_q, and another one representing computing time C_t (i.e., the time to perform computation within one step of the reactive system). Such cost functions must be manually defined, on the basis of e.g., WCET analysis. We propose simple and basic characteristics; they could be made more elaborate, and more realistic in terms of real complex systems: the point here is rather to show how this can be used by optimal discrete control synthesis.

Modes. The functions are defined for each mode of each task: $q_{ij} = C_q(M_{ij})$ and $c_{ij} = C_t(M_{ij})$. One can also think to associate costs to the events in order to take into account the time necessary for mode changes. In this paper, we consider *a priori* that a task in `Idle` or `Wait` state has a null cost and quality. However, a non-null cost of a waiting task could e.g. be used to account for the possible operating system overhead We also consider that the cost and quality of a mode are related in such a way that: $\forall j,k : q_{ij} \geq q_{ik} \Rightarrow c_{ij} \geq c_{ik}$. They need not be considered proportional, though (e.g., computation of inertia in movement control involves a big time cost for a small precision gain, when acceleration is small).

Example 7 *In the example of the three-mode task pattern of Figure 12, we can define costs for two instances T_1 and T_2 as shown in Figure 18.*

Tasks. For a task T_i, we can define its current time cost: $t_i = \sum_j C_t(M_{ij}) * \delta_{ij}$[1] and, accordingly, its current quality: $q_i = \sum_j C_q(M_{ij}) * \delta_{ij}$. Let us recall that, in our framework, only one mode is active at each time. So the current task cost is the cost of the currently active mode.

[1] where δ_{ij} is equal to 1 whenever the task i is in mode M_{ij}, 0 otherwise.

		I_i	W_i	H_i	M_i	L_i
T_1	C_t	0	0	7	5	2
	C_q	0	0	3	2	1
T_2	C_t	0	0	6	3	1
	C_q	0	0	6	4	3

Figure 18: Cost and quality weights for modes.

Applications. For a composition of the n tasks, the current global time cost (within a cycle) is: $T = \sum_i t_i$. The current global quality of an application is: $Q = \sum_i q_i$. In the example of the two instances of the three-mode task pattern, and for the particular configurations illustrated in Figure 14, they are characterized as shown in Figure 19.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
T	5	10	8	5	13	11	8
Q	6	7	6	5	9	8	7

Figure 19: Global cost and quality weights for the application.

Alternately, and depending on the notion of quality adopted, we could take it to be the average of local qualities, as in: $Q_{alt} = \left(\frac{\sum_i q_i}{n}\right)$ but it would be quite like the sum if you consider that you want to use it for bounding or maximization purposes, as we will see next. Also, in our approach, it allows for the inactive tasks not to interfere in the value of quality. Other notions of quality would require another model, like the minimum, for cases where the global quality is that of the weakest component.

3.4.2 Associated properties and optimization

Bounding the sum of costs. Sharing the processor means that at each cycle, the time needed to compute each of the tasks (one step of each) must be contained within a global period T_{max}, i.e.: *the sum of local costs should always be less than a given maximum* $T < T_{max}$. The control of mode switching must *go only to global configurations where this property is true.* This is a logical invariance objective.

In the example, we take $T_{max} = 11$. Then, we have to exclude configurations S_5 and S_6, i.e., the controller has to *forbid* the transitions from S_1 to S_5 and S_6.

Maximizing quality. We want to deliver the functionality at the best possible global quality (i.e., least degradation): maximal, and evenly distributed. From a configuration S with successors $S' \in Succ(S)$, we want to keep only transitions going to a configuration such that the property is satisfied.

First objective: maximizing global quality. *Amongst the remaining possible next global configurations* S'*, go only to those where the sum of local qualities is maximal* i.e., $Q = Q_{max} = max_{S' \in Succ(S)}(Q(S'))$. In the example, on the remaining successor states, we have $Q_{max} = 7$: we keep only configurations S_2 and S_7. There can be several successors with equal, maximal, global quality, but with different local distributions amongst modes.

Second objective: minimize time. For an equivalent quality, we want to pay the least cost: *Amongst the remaining possible next global configurations S', go only to those where the time cost T is minimum, i.e., $T = T_{min} = min_{S' \in Succ(S)}(T(S))$.* In the example, for the same quality $Q_{max} = 7$, S_2 has a time cost of 10, S_7 has a time cost of 8. Hence, we keep only configuration S_7. That is to say, the controller must authorize only ($\texttt{c2}_1$,ϵ) from S_1.

Alternate objective: having a homogeneous quality. Another objective could concern homogeneous quality, defined as: amongst configurations with equal maximal quality, choose those where the values are closest to the average. That is to say, the property we want to be satisfied is: *amongst the remaining possible next global configurations S', go only to those where the difference D between local qualities and the average is minimum, i.e., $D = D_{min} = min_{S' \in Succ(S)}(D(S))$* . The average quality of modes *of active tasks* (which are m in number) is: $Q_{avg} = \frac{Q}{m}$ We define the difference $D(S) = \sum_i (|q_i - Q_{avg}| \times a_i)$. More elaborate notions of distance could be meaningful here, like variance (the average of distances to the average) or standard deviation. In the example, on the states remaining after the first objective, we have $Q_{avg} = 3\ .\ 5$, and $D(S_2) = 1, D(S_7) = 5$. Hence, we keep only configuration S_2. That is to say, the controller must authorize only ($\texttt{c0}_1$ `and` $\texttt{c1}_2$) from S_1.

3.4.3 Other possible interpretations

We can also have other interpretations of weights, like: degradation of quality implies a longer time needed for the same result (we can then apply optimization along trajectories, among several ones joining same start and goal). A lower cost in energy can require more time for the functionality to be fulfilled (this goes well with our previous scheme if it is interpreted as a higher quality). We could also consider algorithms to which an *a priori* cost (e.g., search depth) can be associated, according to the time available (as with Taylor function developments, "anytime algorithms"). Also, it would be of interest to consider weights associated to transitions and states and their composition along traces, in order to have optimal synthesis on trajectories.

4 Application: case study of a robotic system

4.1 The robotic system

4.1.1 The components of the system

The system decomposes into sub-systems, according to the actuators: the articulated arm, and the gripper held at the end of the arm. This system could itself become a sub-system, e.g. in an excavation system, completed with the rotating cabin on which the arm is mounted, and the mobile base, carrying the whole, itself composed of two tracks [32, 24].

Each subsystem i.e., actuator, is equipped with a library of control tasks, corresponding to different functionalities of the device, and different ways of achieving them, according to different criteria. The complete system is simply constituted by the composition of all its actuators, each with its control tasks.

The gripper is equipped with 3 tasks.

- The manipulation task `Gmanu` is a standard task. At the lower level, the purpose of this task is to either open or close the grip. It is manually controlled i.e., the operator directly decides on the movements of the grip through teleoperation.

- The task `Gmaint` is the default task. It maintains the gripper at the current opening position.
- The task `Gauto` is an two-modes task. It corresponds to an automated movement control task (e.g. according to force feedback from sensors measuring the grip strength). Its two modes do this according to two versions, differing by the algorithms: the high mode `H` makes more accurate computations of control, involving e.g., mechanical models of friction, whereas the low mode makes an approximation. Hence, the low mode `L` has a lower quality of control, and a lower cost in computation time.

The articulated arm is equipped with 4 tasks.

- The manipulation task `Amanu` is a standard task. It offers manual control to the operator e.g., with possibly sharing of degrees of freedom with an automated control law in computer assisted teleoperation.
- The task `Ahome` is a sustainable task. It brings the arm from whatever position it is in, towards a predefined resting position (e.g., folded in a position where a mechanical brake or blocking device can be put in place). When arrived at this position, it can sustained, thereby keeping the device under control.
- The automated movement task `Auto` is a two-modes task. It is defined by a control law, for example, trajectory following, or sensor-based movement, with the sensor placed at the end of the arm. Its two modes do this according to two versions, differing by the algorithms: the high mode `H` makes more accurate computations of control, involving e.g., mechanical models of inertia (gravity, Coriolis forces, ...) as well as camera-based sensing involving image processing, whereas the low mode makes an approximation, and uses less elaborate sensing. Hence, the low mode `L` has a lower quality of control, and a lower cost in computation time.
- The task `Amaint`, the default task, maintains the current position. It constitutes an actual control task because just cutting off power might result in the arm falling down due to gravity or moving arbitrarily due to strong wind or water current.

4.1.2 The model of the system

Each task is modeled by an instance of the automaton encoding the appropriate task pattern, as introduced above. Each subsystem is described by means of a composition of its tasks, sharing inputs and outputs. The composition of subsystems simply defines the whole system. Hence it describes, at that abstraction level, all possible dynamical behaviors of the robotic system.

This system can be considered for a given application; an example of application is a sequence of bringing the arm in its home position (in order to reinitialize all relevant parameters), then having an automated movement towards some position, then performing a manual arm movement e.g., to finely approach something to be gripped, then manually gripping the object there, and then, finally, bringing the arm back in its home position again.

4.2 Properties and objectives

The model includes amongst its configurations some that are undesirable, for example for reasons of resources to be shared (e.g., an actuator between different control laws), or criteria related to the functionality fulfilled by the tasks (e.g., incompatible side effects on the device or its environment). Amongst the paths described by sequences of transitions, there can also be undesirable ones, for

example for reasons of necessary transitory modes between some tasks (e.g., between velocity-based and position-based movement control of a motor). In order to control these situations, we have to specify the properties on the states and events to be either achieved or avoided. Then, using them as synthesis objective, we can obtain the discrete controller, if it exists, which will constrain the behaviors in such a way that only those satisfying the properties will be allowed.

4.2.1 Generic properties

Independently of the environment within which it appears (surrounding architecture it is embedded in, specific application it is used for), each subsystem can have constraints.

Logical properties. The most basic ones concern presence and uniqueness of control. Moreover, if the automated mission controlling the system does not specify all the controls of all the actuators at all times, it can be useful to specify that in case no other control is active, then the default control law should be activated. Hence, for each actuator,

1. *It must always be under control (otherwise unpredictable movements can occur).*
2. *There must be at most one active control law (otherwise control can incoherent).*

This translates into a synthesis obejctive as seen in Section 3.3.2.

4.2.2 Architecture-specific properties

Logical properties. The architecture grouping the actuators and at the lower level al the corresponding tasks brings possible interactions between different subsystems upon which some properties have to hold. These properties can be static properties, in the sense that they concern properties on states, or dynamic properties on successions of events. Interesting properties to ensure in our example are the following:

1. *No manual manipulation of the gripper when the arm is manually teleoperated.* This corresponds to a safety requirement related to attention span of the human operator, and the avoidance of human failure.

   This translates into the synthesis obejctive of making invariantly false the conjunction of the activity of `Amanu` and `Gmanu`.

2. *between a manual arm movement and an automated one, the default task maintaining the current position should be activated.* This corresponds to a fine recalibration of sensors and actuators between these two movements.

   This translates into using the observer of Section 3.3.3 shown in Figure 17. This is a case where imposing t_2 between t_1 and t_3 can then be seen as avoiding t_3 between t_1 and t_2. Task `Amanu` is t_1, `Aauto` is t_3, and `Amaint` is t_2.

Quantitative properties. For a given architecture, bounds must be respected regarding consumption, e.g., of energy or of computing power. Costs of computing power for each task are assigned as in the table shown in Figure 20.
These costs add up when tasks are active in parallel, thereby defining a global cost. Giving the processor capacity, the upper bound is assumed to be equal to 12. It entails that the subset of states where the global cost is strictly higher than the bound has to be forbidden by control.

Gripper	cost	quality
Gmanu	5	4
Gmaint	3	3
Gauto H	4	5
Gauto L	2	2

Arm	cost	quality
Aauto H	8	8
Aauto L	6	6
Ahome	4	4
Amanu	7	7
Amaint	4	4

Figure 20: cost and quality weights for the tasks on the example.

4.2.3 Application-specific properties

They can be considered according to the case under design. In the example of sequential application above, a relevant property is that the termination of the sequence should always be reachable. A counter-example would be that other properties, quantitative or logical, would involve control sequences going into loops or waiting configurations with no possibility to proceed.

4.3 Model specification in Mode Automata

Tasks. Examples of tasks are given in Figure 21. They are the concrete encodings of the corresponding automata of Figures 10 and 8.

```
AUTOMATON Ahome
STATES
 I_Ahome init
   [ start_Ahome=req_Ahome and go_Ahome;
     end_Ahome=false; ]
 W_Ahome
   [ start_Ahome=go_Ahome; ]
 S_Ahome
   [ end_Ahome=go_Ahome; ]
 A_Ahome
   [ end_Ahome=stop_A and go_Ahome;
     start_Ahome=false; ]

TRANS
 FROM I_Ahome TO W_Ahome
     [ req_Ahome and not go_Ahome ]
 FROM I_Ahome TO A_Ahome
     [ req_Ahome and go_Ahome ]
 FROM W_Ahome TO A_Ahome
     [ go_Ahome ]
 FROM A_Ahome TO S_Ahome
     [ stop_A and not go_Ahome ]
 FROM A_Ahome TO I_Ahome
     [ stop_A and go_Ahome ]
 FROM S_Ahome TO I_Ahome
     [ go_Ahome ]
```

sustainable task Ahome

```
AUTOMATON Amanu

STATES
 I_Amanu        init
   [ start_Amanu=req_Amanu and go_Amanu;
     end_Amanu=false; ]
 W_Amanu
   [ start_Amanu=go_Amanu; ]
 A_Amanu
   [ end_Amanu=stop_A; start_Amanu=false; ]

TRANS
 FROM I_Amanu TO W_Amanu
     [ req_Amanu and not go_Amanu ]
 FROM I_Amanu TO A_Amanu
     [ req_Amanu and go_Amanu ]
 FROM W_Amanu TO A_Amanu
     [ go_Amanu ]
 FROM A_Amanu TO I_Amanu
     [ stop_A ]
```

standard task Amanu

Figure 21: Two tasks in Mode Automata.

The Mode Automata syntax reads as follows. Each automaton is given a name (e.g., `Ahome`), and begins with a declaration of states. The initial state is indicated by the keyword `init`. With

each of them, equations can be associated, defining values of variables (here, we consider Boolean variables), at each instant when the system is in the corresponding state (e.g., in state `S_Ahome`, variable `end_Ahome` takes the value of `go_Ahome`, and in state `I_Ahome` it takes the value `false`).

Then, a list of transitions is given, from source state to sink state, with a condition on variables. For example, from state `S_Ahome`, a transition is taken to state `I_Ahome` if the condition `go_Ahome` is true. One can note that the equations defining `end_Ahome` make that it is true on the instant this transition is taken, and false afterwards; this is a way of encoding in Mode Automata the emission of `end` upon reception of `go` (i.e., `go/end`).

Observer. Figure 22 gives the Mode Automata encoding of the observer from Figure 17, observing that no two activations of automatic arm movement (tasks 1 and 2) occur without a position maintaining task (task 3) in between. The equations in the different states (all the same) define the local variables conditioning the transitions, in terms of the tasks under consideration.

```
AUTOMATON obs
STATES
        Clear   init  [ start1=start_Amanu; end1=end_Amanu;
     start2=start_Amaint; end2=end_Amaint; start3=start_Aauto; end3=end_Aauto; ]
        InT1          [ start1=start_Amanu; end1=end_Amanu;
     start2=start_Amaint; end2=end_Amaint; start3=start_Aauto; end3=end_Aauto; ]
        AfterT1         [ start1=start_Amanu; end1=end_Amanu;
     start2=start_Amaint; end2=end_Amaint; start3=start_Aauto; end3=end_Aauto; ]
        InT2          [ start1=start_Amanu; end1=end_Amanu;
     start2=start_Amaint; end2=end_Amaint; start3=start_Aauto; end3=end_Aauto; ]
        Error          [ start1=start_Amanu; end1=end_Amanu;
     start2=start_Amaint; end2=end_Amaint; start3=start_Aauto; end3=end_Aauto; ]

TRANS
        FROM Clear   TO InT1    WITH rien0 [ start1 ]
        FROM InT1    TO AfterT1 WITH rien0 [ end1 and not start2 and not start3 ]
        FROM InT1    TO InT2    WITH rien0 [ end1 and start2 and not start3 ]
        FROM InT1    TO Error   WITH rien0 [ start3 ]
        FROM AfterT1 TO InT2    WITH rien0 [ start2 and not start3 ]
        FROM AfterT1 TO Error   WITH rien0 [ start3 ]
        FROM InT2    TO Clear   WITH rien0 [ end2 and not start1 ]
        FROM InT2    TO InT1    WITH rien0 [ end2 and start1 ]
        FROM InT2    TO Error   WITH rien0 [ start3 and not end2 ]
```

Figure 22: An observer in Mode Automata.

Application. The application example is encoded as a sequential automaton as in Figure 23. It emits requests to the tasks, and proceeds into sequence upon their termination, sending a request to the next, until the `End` state.

Complete model. The complete model is obtained by assembling task models and observers, by synchronous composition, as shown in Figure 24. The `PAR` and `ENDPAR` is the parallel composition operator, each of the branches being refined by the instruction `RAFF` applied to the corresponding automaton

This complete model is then compiled, according to Figure 7, using MATOU, into an executable format, on the one hand, and on the other hand into the `z3z` format, an equational encoding of the global transition system, taken as input by the verification and synthesis tool SIGALI.

```
AUTOMATON appli
STATES
 Begin init [ req_Ahome = true ; ]
 InAhome1   [ req_Ahome = false ; req_Aauto = end_Ahome ; ]
 InAauto    [ req_Aauto = false ; req_Amanu = end_Aauto ; ]
 InAmanu    [ req_Amanu = false ; req_Gmanu = end_Amanu ; ]
 InGmanu    [ req_Gmanu = false ; req_Ahome = end_Gmanu ; ]
 InAhome2   [ req_Ahome = false ; ]
 End

TRANS
 FROM Begin    TO InAhome1 [ req_Ahome ]
FROM InAhome1 TO InAauto  [ end_Ahome ]
 FROM InAauto  TO InAmanu  [ end_Aauto ]
FROM InAmanu  TO InGmanu  [ end_Amanu ]
 FROM InGmanu  TO InAhome2 [ end_Gmanu ]
FROM InAhome2 TO End      [ end_Ahome ]
```

Figure 23: A sequential application in Mode Automata.

```
PAR
    % observer %  RAFF obs
        % grip %  RAFF Gmanu
                  RAFF Gmaint
                  RAFF Gauto
         % arm %  RAFF Aauto
                  RAFF Ahome
                  RAFF Amanu
                  RAFF Amaint
 % application %  RAFF appli
ENDPAR
```

Figure 24: The complete model in Mode Automata.

4.4 Controller synthesis with SIGALI

Using the SIGALI tool begins with loading some libraries for particular synthesis operations, and loading the `z3z` file generated just above. That file features amongst others a declaration of the system variables: their names are constructed from the names in the mode automata, usually extended with the name of the automaton in which they appear and an integer suffix. A typical example is the `Error` state of the observer `obs`, recognizable in variable `Error_de_obs_0`.

Observer. The observer recognizes sequences of events leading to the `error` state; we want to obtain a controlled system where these sequences are not featured: in other words, we want to *make invariant* the sub-states of states where the observer is not in state `Error`, i.e., where `Error_de_obs_0` is false. This is concretely encoded as follows, by first defining the subset of states called `PROP_obs`, where `B_False` designates the set of states where the variable is false. The synthesis operation for invariance, called `S_Security` is then applied to the original system `S` with the set `PROP_obs`, the result redefining `S`.

```
PROP_obs :B_False(S, Error_de_obs_0) ;
S : S_Security(S, PROP_obs);
```

Unicity of control. Every actuator should be under control i.e., at least one control task should active, and at most one. The first part is obtained as follows, by giving an expression for one active task, for each actuator.

```
One_G : A_Gmanu_de_Gmanu_1 or A_Gmaint_de_Gmaint_2 or AH_Gauto_de_Gauto_3
          or AL_Gauto_de_Gauto_3 ;
One_A : AH_Aauto_de_Aauto_4 or AL_Aauto_de_Aauto_4 or A_Ahome_de_Ahome_5
          or S_Ahome_de_Ahome_5 or A_Amanu_de_Amanu_6   or A_Amaint_de_Amaint_7 ;
One : One_G and One_A  ;
```

The same can be done for the second part, by defining a variable `Several` true when several tasks are active on either actuator. The controller should then make invariant the set where `One` is true and `Several` is false, which is encoded as follows:

```
PROP_one:B_True(S, One);
PROP_only:B_False(S, Several) ;
PROP_one_only : intersection(PROP_one,PROP_only) ;

S : S_Security(S, PROP_one_only);
```

Insuring exclusion between manual control for the gripper and for the arm goes along the same lines:

```
PROP_excl : B_False(S, A_Gmanu_de_Gmanu_1 and A_Amanu_de_Amanu_6) ;
S : S_Security(S, PROP_excl);
```

where an expression on state variables is made invariantly false.

Reachability. A resting position of the system is defined, where both actuators are controlled by position maintaining tasks. This configuration must be always reachable, from any other configuration of the system.

```
Rest : I_Gmanu_de_Gmanu_1 and A_Gmaint_de_Gmaint_2 and I_Gauto_de_Gauto_3
       and
       I_Aauto_de_Aauto_4 and I_Ahome_de_Ahome_5 and I_Amanu_de_Amanu_6
            and A_Amaint_de_Amaint_7 ;

P_Rest  : B_True(S, Rest);

S : S_Reachable(S, P_Rest);
```

This objective is to be applied after all the invariance ones, as the removal of transitions might break the reachability. In the case of the systems we consider, actually, reachability can be treated by verification, at the end of the synthesis process, as if it is not satisfied, then it is unlikely that it can be controlled.

Another reachability objective can be associated with the sequential application: the reachability of its terminal state `End`:

```
P_End   : B_True(S, End_de_appli_8);

S : S_Reachable(S, P_End);
```

Weight-related properties. These properties involve declaring values of the computing power consumption, and specifying the bound.

Declaring weights. This is done as follows for each state variables corresponding to an active state: using the SIGALI instruction `a_var`, a value is associated with the variable having the value true; for all other cases, it is given the value 0, which is neutral for addition. This way, combining values of composed task controllers is done simply by adding their costs into `C_G` for the gripper, and `C` for the global cost.

```
C_GHauto        :  a_var(AH_Gauto_de_Gauto_3,   0 , 4 ,0) ;
C_GLauto        :  a_var(AL_Gauto_de_Gauto_3,   0 , 2 ,0) ;

C_G     :  C_Gmanu + C_Gmaint + C_GHauto + C_GLauto ;

C       :  C_G + C_A ;
```

Respecting bounds. For this, a bound is declared, and the instruction `a_inf` is used to obtain the set of states `InBound_power` where the cost in lower than the bound. The latter is then being made invariant.

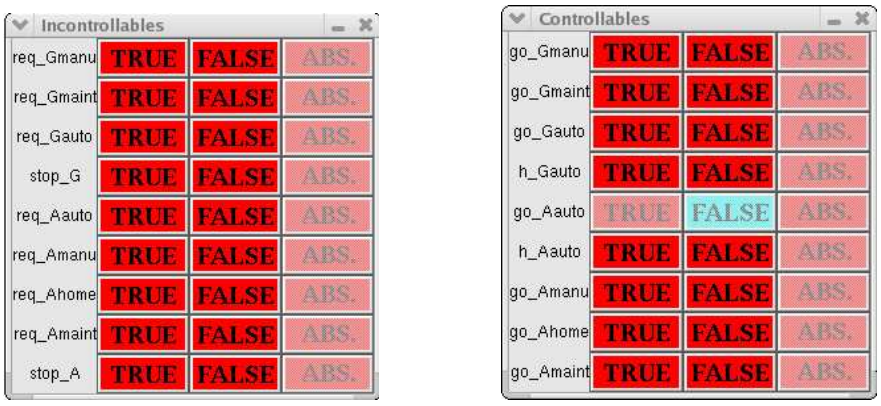
```
Bound_power     : 12 ;

InBound_power : a_inf(C, Bound_power) ;

S : S_Security(S, InBound_power) ;
```

4.5 Interactive simulation

At every phase in the above construction of the model and control, it is possible to obtain a simulator of the behaviors of the controlled system. This way, a user can observe how the behaviors change when adding one objective, and verify whether the constraints added correspond to the problem to be solved. A simulation consists of iterating, step by step, the following three operations:

1. *simulating the environment* is done through the uncontrollable inputs panel (see Fig. 25(a)), where one can enter the requests from the operator, and the events signaling termination of tasks.

2. *choosing among correct controls* is done through the controllable events panel (see Fig. 25(b)). Values ruled out by constraints are represented by non-selectable buttons. There can be possibly several allowed values, if the constraints do not completely determine the control from the inputs. In order to obtain a control function, i.e., an input-deterministic controller, the specification has to be strengthened, or optimization criteria (w.r.t. costs on states or events) have to be applied, or a random choice can be applied. Then, the control of the system is directed in a shared way by, partly, requests from the user, and partly, control by the automatism. This is reminiscent of the concept of teleoperation, where movements of an actuator are directed partially by a human operator, with a sharing of some degrees of freedom with automated control laws. As such, this mode of interactive control can be called discrete teleoperation.



(a) The uncontrollable inputs panel (b) The controllable inputs panel

Figure 25: Panels for interactive simulation.

In Figure 25(b), the situation shown is related to our example, where the request for `Aauto` is refused, because there has been no `Amaint` since its last activation.

3. *the dynamical evolution* is observed with the task states display.. Changes in behaviors obtained after adding a control objective can show in, e.g., observing that a requested task goes into wait state when another, incompatible one is active, until termination.

5 Conclusions and perspectives

5.1 Results

We have proposed modelling patterns for multi-tasks real-time control systems, such as in robotic, automotive or avionic systems, as well as for some of their relevant safety properties, that have to be insured through all possible switchings in control activities. This formal modelling in terms of transition systems allows for the application of the discrete controller synthesis technique. This provides us with a systematic methodology, for the automatic generation of safe task handlers which enforce the properties given as objectives. We experimented the approach using models, languages and tools from the synchronous approach to reactive systems.

5.2 Future work

Possible extensions based on these results are as follows:

- defining an end-user-friendly, domain-specific programming language on these bases, where languages constructs are in terms of tasks, modes, start and end control, resources and their shareability, and a specification of sequences of tasks constituting an application; this will alleviate all need for direct formal technical expertise for the end-user.
- exploiting more advanced synthesis techniques, like optimization on bounded-length paths, especially when applied after a reachability objective, or hierarchical structure in synthesis [22, 9], ...

- exploring execution mechanisms, replacing simulation by coupling the resulting controller with an exploitation system hosting the actual tasks;
- exploring other application domains, where a specific use fo the same kind of techniques can be made, e.g., fault tolerance [7, 10].

Acknowledgments. We gratefuly acknowledge helpful discussions with and influences by M. Abdennebi, H. Alla, K. Altisen, A. Clodic, S. Iddir, P. Manon, F. Maraninchi, A. Medina-Rodriguez, T. Neveu, P. P. Parida.

References

[1] A. Aziz, F. Balarin, R. K. Brayton, M. D. Dibenedetto, A. Sladanha, and A. L. Sangiovanni-Vincentelli. Supervisory control of finite state machines. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 279–292, Liege, Belgium, July 1995. Springer Verlag.

[2] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the European Symposium on Programming, ESOP'03,* April 7 - 11, 2003,Warsaw, Poland, 2003.

[3] K. Altisen, G. Gößler, and J. Sifakis. Scheduler modelling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23(1), 2002.

[4] B. Bérard et al. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[5] J-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The ORCCAD architecture. *Int. J. of Robotics Research*, 17(4), 1998.

[6] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

[7] E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *Proceedings of the 7th Workshop on Discrete Event Systems, WODES'04,* September 22-24, 2004, Reims, France, 2004.

[8] G. Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE Transactions on Information and Systems*, E76-D(11):1333–1340, November 1993.

[9] B. Gaudin and H Marchand. Supervisory control of product and hierarchical discrete event systems. *European Journal of Control*, 10(2), 2004.

[10] Alain Girault and Éric Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems, FMICS 04,* September 20-21, 2004, Linz, Austria, pages 125–142 (Tech. Rep. SEA–SR–03, Inst. for Sys. Eng. & Automation, Kepler University Linz), 2004.

[11] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[12] N. Halbwachs. Synchronous programming of reactive systems – a tutorial and commented bibliography. In *Proceedings of the International Conference on Computer-Aided Verification, CAV'98,* Vancouver, Canada, june 1998, number 1427 in Lecture Notes in Computer science (LNCS). Springer-Verlag, 1998. LNCS nr. 1427.

[13] D. Harel and A. Naamad. The STATEMATE semantics of STATECHARTS. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.

[14] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In *Proceedings of the 6th International SPIN Workshop on Practical Aspects of Model Checking, Toulouse, France, September 1999*, number 1680 in Lecture Notes in Computer Science (LNCS), pages 92–107. Springer Verlag, 1999.

[15] Ch. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *5th Euromicro Conference on Real-Time Systems (ECRTS'03), Porto, Portugal, July*, 2003.

[16] O. Kushnarenko and S. S. Pinchinat. Intensional approachs for symbolic methods. In *Electronic Notes in Theoretical Computer Science*, volume 18, 1998.

[17] P. Le Guernic. Compilation involving model-checking and controller synthesis. Personal communication, 1996.

[18] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E. W. Mayr and C. Puech, editors, *Proceedings STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer-Verlag, 1995.

[19] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[20] F. Maraninchi, Y. Rémond, and E. Rutten. Effective programming language support for discrete-continuous mode-switching control systems. In *Proc. of the 40th IEEE Conf. on Decision and Control, CDC'01,* Orlando, Florida, dec, 2001.

[21] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamical System: Theory and Applications*, 10(4), October 2000.

[22] H. Marchand and B. Gaudin. Supervisory control problems of hierarchical finite state machines. In *Proc. of the 41th IEEE Conference on Decision and Control, CDC'02,* Las Vegas, USA, dec, 2002.

[23] H. Marchand and M. Le Borgne. The supervisory control problem of discrete event systems using polynomial methods. Research Report 1271, Irisa, October 1999.

[24] H. Marchand and E Rutten. A case study in discrete control synthesis for excavator operation. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, SMC'2002,* October 6-9, 2002, Hammamet, Tunisia, 2002.

[25] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems, ECRTS'02,* June 19th - 21th, 2002, Vienna, Austria, pages 241–248, 2002.

[26] S. Pinchinat and H. Marchand. Symbolic abstractions of automata. In *Proc of 5th Workshop on Discrete Event Systems, WODES 2000*, pages 39–48, Ghent, Belgium, August 2000.

[27] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[28] J. Real and A. Wellings. Implementing mode changes with shared resources in ada. In *Proceedings of the* 11th Euromicro Conference on Real-Time Systems, ECRTS'99*, York, England, UK, June 9th - 11th*, 1999.

[29] I. Romanovski and P. Caines. Multi-agent product systems: analysis, synthesis and control. In *Proc of 6th Workshop on Discrete Event Systems, WODES 2002*, Zaragoza, Spain, October 2002.

[30] E. Rutten. A framework for using discrete control synthesis in safe robotic programming and teleoperation. In *Proc. IEEE Int. Conf. on Robotics and Automation, ICRA'2001,* Seoul, Korea, may, 2001.

[31] E. Rutten and H. Marchand. Task-level programming for control systems using discrete control synthesis. Rapport de Recherche 4389, INRIA, February 2002. www.inria.fr/rrrt/rr-4389.html.

[32] D. Simon, M. Personnaz, and R. Horaud. TELEDIMOS: telepresence simulation platform for civil work machines: real-time simulation and 3d vision reconstruction. In *Proc. IARP Workshop on Advances in Robotics for Mining and Underground Applications, Brisbane, Australia, Oct.*, 2000.

[33] S. Takai and T. Ushio. Supervisory control of a class of concurrent discrete event systems. *IEICE Transactions on Fundamentals*, E87-A(4):850–855, 2004.

[34] L. Nana Tchamnda, J.L. Fleureau, and L. Marcé. A control system for pilot : software architecture and implemantation issues. In *Proceedings of ANS'01, ANS 9th International Topical Meeting on Robotics and Remote Systems, Seattle, Washington*, March 2001.

[35] K. Tindell, A. Burns, and A. Wellings. Mode changes in priority pre-emptive scheduled systems. In *Proceedings of the IEEE Real Time Systems Symposium, Phoenix, Arizona*, pages 100–109, December 1992.

[36] N. Turro, E. Coste-Manière, and O. Khatib. A portable programming framework. In *Experimental Robotics VI*, number 250 in LNCIS. Springer Verlag, 2000.

Contents

List of Figures

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399